LAB NOTES

Accessing the Windows API from the DOS Box, Part 1

BY ANDREW SCHULMAN

Why can't you start a Windows application from Windows' own DOS box? Why should you need a special utility such as WINSTART, by Douglas Boling (Utilities, June 30, 1992)? The answer is that WinExec()—the function that knows how to run Windows programs—is part of the Windows API and, strangely enough, the Windows API is not available to DOS programs running under Windows. The DOS box in Windows *really is* just a box: It seems to have almost no connection with Windows.

Well, then, why *would* you want to run a Windows program from the DOS box or call Windows API functions from a DOS program?

Actually, the very inconvenience and confusion caused by the unavailability of WinExec() in the DOS box shows why the DOS box should be not just *in* Windows, but *of* Windows. The need to remember whether you're in DOS "mode" or Windows "mode," and which kinds of programs you have, and which side of the fence they need to run on, is really part of the old-school way of doing things that Windows supposedly seeks to replace.

An analogy might perhaps be the start of a better answer. As matters stand, it's almost as if Windows were a *very* local area network, with the DOS box running on one machine and the rest of Windows running on another machine. In fact, this is precisely how Windows Enhanced mode is implemented, using a feature of the 80386 that creates a separate *virtual machine* for each DOS box and another for the Windows core. Each "machine" has its own separate address space, and not much communication goes on between them.

But wait a minute. We know that on a network machines *can* still communicate with each other, even though they are separate entities without a shared address space. Likewise, it turns out that there *is* a way to get at some Windows functionality from a plain-vanilla DOS program. Although it often appears otherwise, the DOS box is not completely isolated from the rest of Windows. Windows provides a small set of interrupt-based services to DOS programs, and while far from ideal, these services are sufficient to form a bridge between the DOS box and the rest of Windows.

With an understanding of WINOLDAP and the Clipboard, you can give DOS programs access to Windows functions.

From a programmer's perspective, there are several APIs that Windows provides to DOS programs. The best known is the DOS Protected-Mode Interface (DPMI). Unlike the standard Windows API, which relies on DLLs (dynamic link libraries), DPMI uses software interrupts accessible to DOS programs. Specifically, DPMI uses INT 31h and the *multiplex* interrupt, INT 2Fh. And Windows provides several non-DPMI INT 2Fh functions as well.

This three-part article will discuss a small set of INT 2Fh functions that Windows provides for getting data in and out of the Windows Clipboard from a DOS program. Douglas Boling briefly discussed these functions at the end of his article "DOSCLIP.EXE: A Clipboard for All Sessions" (Utilities, April 14, 1992). In these Lab Notes columns I'll cover them in much greater detail. Part 1 uses these functions to build a DOS library of Windows Clipboard-access functions, and then uses these higher-level functions to create several small DOS-based Clipboard-access utilities.

In the next two issues, I'll show how to build on these functions to access other Windows API functions, including WinExec(). The basic idea is that, while DOS programs can't directly call Windows API functions such as WinExec(), DOS applications can use the Clipboard to pass messages to a Windows program that *can* call the API functions. Under this scheme, the Windows program acts as an agent, server, or surrogate that carries out actions on behalf of its less-capable DOS client. The Windows Clipboard will thus serve as our "transport layer" in this very local area network. The conclusion to this series will consider other, perhaps more suitable "transport layers" whose usefulness lies in the fact that, at a sufficiently low level, DOS boxes and Windows all do share the same memory address space.

This article may seem somewhat odd because, while very much about Windows, it is illustrated primarily with DOS programs. For someone who has been doing Windows programming steadily for some time now, this is a welcome change of pace. It is also one of the best reasons for making the Windows API accessible to DOS programs running under Windows: DOS programs are much easier to write! And, quite frankly, not every

program requires (or deserves) to be rewritten as a Windows application. Yet Windows is—or could be—a fine environment in which to run DOS applications.

BETTER THAN NOTHING The Windows functions we'll be calling from DOS are part of the WINOLDAP.MOD Clipboard API. WINOLDAP is a Windows executable responsible for running "old" (that is, DOS) applications. Actually, it's one of several Windows programs that together perform a complicated, baroque dance to run a DOS application. Other participants include the SHELL virtual device driver (VxD), the Virtual Display Driver (VDD), and the "Grabber."

The WINOLDAP Clipboard API gives DOS programs *programmatic* access to the Windows Clipboard. In other words, instead of waiting for a user to mark, cut, copy, and paste some text from the DOS box screen, your program can directly make WINOLDAP open the Windows Clipboard, get text from it, empty it, put data in it, and close it. The program does this using INT 2Fh, with the AH register set to 17h and with the AL register set to various subfunction numbers, as follows:

```
Øh      GetWinOldApVersion()
1h      OpenClipboard()
2h      EmptyClipboard()
3h      SetClipboardData()
4h      GetClipboardDataSize()
5h      GetClipboardData()
8h      CloseClipboard()
9h      CompactClipboard()
ØAh     GetDeviceCapabilities()
```

We'll get to the details about these functions a little later, but here it is important to note that the WINOLDAP Clipboard API is available in Windows Enhanced mode only. This makes sense, because when you run a DOS application in Standard mode, most of Windows is swapped out to disk and is consequently unavailable.

In Enhanced mode, WINOLDAP is actually called WINOA386; it works closely with the SHELL VxD built into WIN386.EXE. (Please note that SHELL VxD must not be confused with *SHELL.DLL* in Windows 3.1.) When a DOS program issues an INT 2Fh Clipboard API call, the call is originally caught by the SHELL VxD, which uses the Call_Priority_VM_Event function to send the request off to WINOA386. WINOA386 contains a table of functions that carry out requests such as these; for example, it uses the Windows API call OpenClipboard() to satisfy an INT 2Fh AX=1701h.

While these INT 2Fh functions are similar to the "native" Windows Clipboard API calls documented in the *Microsoft Windows Programmer's Reference,* they have been adjusted for the fact that they are being called—through many, many levels of indirection—in a "remote" fashion, almost as if from another machine.

The WINOLDAP Clipboard API is important because it provides the only direct access that a DOS program has to the Windows side of the fence.

For example, the Windows API function SetClipboardData() expects a Windows global memory handle, which is the last thing a DOS program is likely to have. Thus, the implementation for INT 2Fh AX=1703h does all the work of allocating a global memory handle and copying the data into it before calling SetClipboardData(). For bitmaps, it calls CreateBitmap(), another thing a DOS program unfortunately can't do for itself. Also, all pointers in the bitmap data structure have been flattened into actual data.

Likewise, there is no such Windows API function as GetClipboardDataSize(). If a Windows application wants to find the size of an object from the Clipboard, it calls the GlobalSize() function. But since DOS applications don't have direct access to GlobalSize(), WINOA386 provides the GetClipboardDataSize() function. The function's implementation calls GetClipboardData() and then GlobalSize(), thus acting as a surrogate function-caller for the DOS program. In this particular model, it really wouldn't have been too difficult to give DOS programs remote access to the entire Windows API!

The API was fully documented in the Windows 2.*x* SDK (Software Development Kit), but it was omitted from the 3.0 SDK. If you don't happen to have SDK manuals from 1987 sitting in your basement along with back copies of *National Geographic* and *PC Magazine,* you can get full details on the API from a Microsoft KnowledgeBase article, "Access to the Windows Clipboard by DOS Applications" (September 20, 1991; document Q67675). Like all such articles, this one is available from the excellent Microsoft KnowledgeBase on CompuServe (type GO MSKB at the prompt).

Even if you're not particularly interested in the Windows Clipboard as such, the WINOLDAP Clipboard API is still important because it provides the only direct access that a DOS program has to the Windows side of the fence. While intended for the transmission of data such as text and graphics, we can build on the Clipboard API to create a general-purpose "gateway" from the DOS box to the rest of Windows. Once we can move text from a running DOS program into Windows, we can move commands and even make Windows API calls. Limited as it is, the Clipboard API is one of the few ways Microsoft has provided to open Windows from the DOS box.

MAKING THE API LIBRARY Details of the nine functions that make up the WINOLDAP Clipboard API are shown in Figure 1. Unlike the DLL-based Windows API provided to bona fide Windows programs, this Clipboard API expects a software interrupt (INT 2Fh) with parameters in CPU registers.

How do we take the information in Figure 1 and turn it into something that we use in a DOS program? Suppose that we want to open the Windows Clipboard. (You can't do much of anything to the Clipboard without first opening it.) Using assembly language, the information provided for OpenClipboard() in Figure 1 becomes the code shown in Figure 2.

This code assumes, of course, that GetWinOldApVersion() has already been called to determine that the WINOLDAP Clipboard API is available. If it

Windows Clipboard INT 2Fh API

```
GetWinOldApVersion()
    AX = 1700h
  Return:
    AX == 1700h: Clipboard functions not available (Windows not running, or is running in Standard mode)
    AX <> 1700h: AL.AH = WINOLDAP (not Windows!) Version number (Major.minor)

OpenClipboard()
    AX = 1701h
  Return:
    AX <> 0: Success (Clipboard opened)
    AX == 0: Failure (cannot open Clipboard; already opened by another program)
  Note:
    OpenClipboard() does not return a handle to the Clipboard. It is simply an operation that must be performed
    before any other Clipboard-access routines.

EmptyClipboard()
    AX = 1702h
  Return:
    AX <> 0: Success (Clipboard emptied of previous contents)
    AX == 0: Failure

SetClipboardData()
    AX = 1703h
    DX = Clipboard format; WINOLDAP supports the following (CF_ stands for "Clipboard format"):
         CF_TEXT          1
         CF_BITMAP        2
         CF_OEMTEXT       7
         CF_DSPTEXT       81h (owner display)
         CF_DSPBITMAP     82h (owner display)
    ES:BX -> Far pointer to data
    SI:CX = Size of data in bytes (can be > 64k)
  Return:
    AX <> 0: Success (data copied into Clipboard)
    AX == 0: Failure
  Notes:
    (1) The DOS application should call CompactClipboard() (see below) before calling SetClipboardData() to determine
    if there is sufficient memory.
    (2) The bitmap formats mimic the actual Windows bitmap structure, except that instead of including a handle
    or pointer to other memory containing the actual data, the data immediately follows the structure, which thus
    acts as a header prefacing the data.
    (3) Other formats can be exchanged between a DOS application and Windows (even formats produced with
    RegisterClipboard-Format), but these are the ones documented by Microsoft for use by DOS programs.

GetClipboardDataSize()
    AX = 1704h
    DX = Clipboard format (see SetClipboardData(), above)
  Return:
    DX:AX <> 0: Size of the data in bytes, including any headers
    DX:AX == 0: Clipboard does not contain data in the specified format

GetClipboardData()
    AX = 1705h
    DX = Clipboard format (see SetClipboardData(), above)
    ES:BX -> Far pointer to buffer to hold data
  Return:
    AX <> 0: Success; buffer pointed at by ES:BX now contains data
    AX == 0: Failure (for example, Clipboard does not contain data in the specified format)
  Note:
    To determine the size of the buffer pointed to by ES:BX, the DOS program must call GetClipboardDataSize();
    WINOLDAP does not know the size of your buffer. Call GetClipboardDataSize() just before calling
    GetClipboardData().

CloseClipboard()
    AX = 1708h
  Returns:
    AX <> 0: Success
    AX == 0: Failure (could not close Clipboard)
  Note:
    The DOS program must close the Clipboard before exiting, or it will disable Clipboard access for all other programs
    (both DOS and Windows programs).

CompactClipboard()
    AX = 1709h
    SI:CX = Required memory size in bytes
  Returns:
    DX:AX <> 0: Number of bytes in largest block of free memory
    DX:AX == 0: Failure (insufficient memory)

GetDeviceCaps()
    AX = 170Ah
    DX = GDI information index (see WINDOWS.H #defines for GetDeviceCaps(); for example, HORZSIZE = 4 and
    VERTSIZE = 6)
  Returns:
    AX <> 0: Value of desired item
    AX == 0: Error
  Note:
    In the Windows API, GetDeviceCaps() takes an hDC (handle to display context) parameter; from a DOS program,
    the hDC is implied. The device capabilities ("devicecaps") are useful for DOS programs that want to display
    bitmaps.
```

Figure 1: **The nine functions that comprise the Clipboard API give DOS programs interrupt-driven access to Windows.**

isn't (that is, if Windows isn't running, or if it's running only in Standard mode) it isn't even valid to call any of the other functions, including OpenClipboard(). The point here, however, is simply to illustrate how Figure 1 turns into actual code.

The problem with the code shown in Figure 2 is that writing in assembly language is just plain tedious. Even if we were writing in assembly language, we wouldn't want INT 2Fh calls sprinkled all over our code. So, no matter what, we must have a library that makes it easier to call the Windows Clipboard functions from a DOS program.

In a C program, for example, we would want to be able to write code such as this:

```
if (! OpenClipboard())
    return 0;
if (! EmptyClipboard())
 {
    CloseClipboard();
    return 0;
 }
 // etc.
```

To implement such a C-callable function library we need a way to set CPU registers and generate software interrupts. Although these capabilities are not normally found in ANSI-standard C, every C compiler for the PC provides them, as do most other programming languages for the PC (including QuickBASIC and Turbo Pascal).

Until recently, most C programs that needed such capabilities used the `int86()`, `int86x()`, `intdos()`, or `intdosx()` functions from DOS.H. However, both Borland C++ and Microsoft C/C++ provide *inline assembler*. Whereas with a function such as `int86x()` you work with an "image" of the CPU registers, with inline assembler you work with the actual CPU registers. This makes it easy to write efficient C functions that map directly onto low-level API calls.

Three examples—OpenClipboard(), CloseClipboard(), and SetClipboardData()—are shown in Figure 3. These functions are straightforward embodiments of the interface shown in Figure 1, save in three respects. First, OpenClipboard() sets a variable named clip_open,

and CloseClipboard() clears it. The clip_open variable thus holds the current "state" of the Clipboard and is used by another part of the library that will be explained below.

Second, it might be unclear just how SetClipboardData() produces a return value, since it does not contain a C *return* statement. The explanation is that functions compiled by C compilers on the PC return 2-byte quantities in AX and 4-byte quantities in DX:AX. Since the return code from the INT 2Fh call is already in AX, it automatically becomes the function's return value.

Finally, note that SetClipboardData() PUSHes and POPs the SI register. This register may be used by the compiler itself, and since the SetClipboardData() interface dictates that we use it, we must save and restore it. Also note that although SetClipboardData() doesn't use it, the DI register may also be used by the compiler.

The three functions in Figure 3 are excerpted from a larger file, WINCLIP.C. Since the implementation of the other functions—GetClipboardData() and so on—can be extrapolated from the examples provided above, they are not shown here. They are available in WINCLIP.C, however, which can be downloaded from Library 3 (Lab Notes) of the Utilities/Tips Forum on PC MagNet. All the files mentioned in this column can be downloaded from PC MagNet, archived as CLIP1.ZIP. If you don't have a modem, they are also available on-disk by mail. Simply send a postcard with your name, address, and preferred disk size to the attention of Katherine West, Utilities, PC Magazine, One Park Ave., New York, NY 10016. No phone calls, please.

WINCLIP.C assumes it is being called from real mode. If you have a protected-mode DOS program, you can use the DPMI interface or your DOS extender's library to produce a protected-mode version of WINCLIP.C. In fact, I originally wrote WINCLIP.C for Phar Lap's 286|DOS-Extender and 386|DOS-Extender and subsequently ported it to real mode. In general, only functions that manipulate segment registers should require alteration for protected mode; in the case of the INT 2Fh Clipboard API (which is real mode only), that means GetClipboardData() and SetClipboardData().

OPENCLIPBOARD()
Partial Listing

```
mov ax, 1701h           ; OpenClipboard function
int 2fh                 ; call WINOLDAP API
or ax, ax               ; is return 0?
jz failure              ; go somewhere else if it is
; clipboard is now open
```

Figure 2: **This code excerpt shows how to call the OpenClipboard() function from assembly language.**

WINCLIP.C
Partial Listing

```
unsigned OpenClipboard(void)
{
    unsigned retval;
    _asm mov ax, 1701h
    _asm int 2fh
    _asm mov retval, ax
    if (retval) clip_open++;  /* see text for explanation */
    return retval;
}

unsigned CloseClipboard(void)
{
    unsigned retval;
    _asm mov ax, 1708h
    _asm int 2fh
    _asm mov retval, ax
    if (retval) clip_open--;  /* see text for explanation */
    Yield();    /* this is a good place to Yield */
    return retval;
}

unsigned SetClipboardData(CF_FORMAT format, unsigned char far *buf,
    unsigned long len)
{
    _asm push si                    /* save away SI; compiler may use it */
    _asm mov ax, 1703h              /* SetClipboardData */
    _asm mov dx, format             /* CF_ format */
    _asm les bx, buf                /* far pointer to buffer */
    _asm mov si, word ptr len+2     /* HIWORD of length */
    _asm mov cx, word ptr len       /* LOWORD of length */
    _asm int 2fh                    /* call WINOLDAP API */
    _asm pop si                     /* restore saved SI */
    /* return value in AX (0 if failure) */
}
```

Figure 3: **C functions for OpenClipboard(), CloseClipboard(), and SetClipboardData(), excerpted from WINCLIP.C.**

To illustrate the necessary changes, a version of SetClipboardData() for 286| DOS-Extender is shown in Figure 4. It uses the Phar Lap API (PHAPI) function DosAllocRealSeg() to allocate conventional memory and the function DosRealIntr() to generate a real-mode interrupt. It is instructive to compare this with the real-mode version of the SetClipboardData() function in Figure 3. Whereas the real-mode SetClipboardData() just stuffs values into registers and makes the INT 2Fh call, the protected-mode version has to jump through a good many hoops in order to access the real-mode API. Specifically, the protected-mode version has to allocate a conventional-memory buffer, copy down into it from protected mode, and then pass a real-mode paragraph address to it.

When we're finished with coding an interface to the API, we need an include file that can be used by programs that want to use the API. WINCLIP.H is shown in Figure 5.

HIGHER-LEVEL SERVICES So far, we've built a C interface library for the Clipboard API; its definitions are in WINCLIP.H and the implementation is in WINCLIP.C. The next step, of course, is to write a sample program to test out the library. (Actually, we should have written the test program first and then produced an interface library that satisfies it, but I've decided to take a more bottom-up approach here.)

SETCLIPBOARDDATA()

Partial Listing

```
unsigned SetClipboardData(CF_FORMAT format, unsigned char far *buf,
    unsigned long len)
{
    REGS16 r;
    unsigned char far *rbuf;
    USHORT seg;
    SEL sel;

    /* allocate conventional memory; get its real-mode paragraph
     address in seg, and its equivalent protected-mode selector
     in sel */
    if (DosAllocRealSeg(len, &seg, &sel) != 0)
        return 0;
    rbuf = MAKEP(sel, 0);
    _fmemcpy(rbuf, buf, (size_t) len); /* use protected-mode selector */

    memset(&r, 0, sizeof(r));
    r.ax = 0x1703;
    r.dx = format;
    r.es = seg;          /* use real-mode paragraph address */
    r.bx = 0;
    r.si = len >> 16;
    r.cx = len & 0xFFFF;
    DosRealIntr(0x2f, &r, 0, 0);
    DosFreeSeg(sel);
    return r.ax;         /* 0 if failure */
}
```

***Figure 4:* This protected-mode version of SetClipboardData() is written for the 286|DOS-Extender.**

TESTCLIP.C, which is shown in Figure 6, is a normal, real-mode DOS program that retrieves text from the Windows Clipboard. It can be compiled with Borland C++, Version 3.0, or Microsoft C, Version 6.0. The program uses plain old `printf()` from the C standard library to display text on `stdout` (DOS standard output); this can be redirected to a file. To get text from the Windows Clipboard, TESTCLIP.C performs the operations of:

- Calling IdentifyWinOldApVersion() to ensure that Windows Enhanced mode is running and that WINOLDAP is available;
- Calling OpenClipboard() to claim temporary ownership of the Clipboard;
- Calling GetClipboardDataSize(CF_TEXT) to find out how many bytes of text (if any) there are;
- Allocating memory to hold the text;
- Calling GetClipboardData() to retrieve it; and
- Calling CloseClipboard() to relinquish ownership of the Clipboard.

You can ensure that TESTCLIP does its job by cutting a file to the Windows Clipboard and then running TESTCLIP with its output redirected to a file. The two files should be identical, save perhaps for an extra newline at the end of the redirected file, which is caused by the C `puts()` function.

In fact, you'll find that it's sometimes a lot easier to pull text out of the Windows Clipboard with TESTCLIP than it is to use a full-blown Windows program. More important, however, is the fact that TESTCLIP also reveals several big problems with the WINOLDAP Clipboard API.

First, there's big trouble if a DOS program succeeds in opening the Windows Clipboard but dies or crashes or exits without closing it. Neither another DOS program nor even a Windows program will be able to access the Clipboard. The only solution is to exit Windows and restart it. And since a DOS application can be terminated simply by hitting Ctrl-C, we need to do something to ensure that the Clipboard gets closed *no matter what.*

TESTCLIP.C does all it can to make sure that it calls CloseClipboard() before exiting. That's the purpose of the clip_open variable. Note, too, that the program calls CloseClipboard() before printing the string with `printf`. That's the point at which the user is most likely to hit Ctrl-C, and in any case we don't need to keep the Clipboard open after we've retrieved the text. Also note that unless you have BREAK=ON, the Ctrl-

WINCLIP.H

Complete Listing

```
/*
WINCLIP.H -- DOS access to Windows Clipboard (Enhanced mode)

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#ifdef METAWARE
/* if using with 32-bit code */
#define far _far
#endif

#ifdef __cplusplus
extern "C" {
#endif

/* higher-level functions */

int WindowsClipboard(void);
int Clipserv(void);
int PutClipStrLen(char *str, unsigned len);
int PutClipString(char *str);
char *GetClipString(void);
void FreeClipString(char *str);

/* lower-level functions */

#define CF_TEXT             1
#define CF_BITMAP           2
#define CF_DSPTEXT          0x81
#define CF_DSPBITMAP        0x82

typedef unsigned short CF_FORMAT;

unsigned long CompactClipboard(unsigned long size);
unsigned CloseClipboard(void);
unsigned EmptyClipboard(void);
unsigned char far *GetClipboardData(CF_FORMAT format,
    unsigned char far *buf);
unsigned long GetClipboardDataSize(CF_FORMAT format);
unsigned GetDeviceCaps(unsigned index);
unsigned IdentifyWinOldApVersion(void);
unsigned OpenClipboard(void);
unsigned SetClipboardData(CF_FORMAT format, unsigned char far *buf,
    unsigned long size);
void Yield(void);

#ifdef __cplusplus
}
#endif
```

***Figure 5:* WINCLIP.H is the include file for use with WINCLIP.**

TESTCLIP.C
Complete Listing

```
/*
TESTCLIP.C -- throw-away test program for Windows clipboard API

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include "winclip.h"

static int clip_open = 0;

void fail(char *s)
{
    if (clip_open)
        CloseClipboard();
    puts(s);
    exit(1);
}

main()
{
    unsigned long len;
    char far *buf;

    if (! IdentifyWinOldApVersion())
        fail("This program must run in a DOS box "
             "under Enhanced mode Windows");

    /* MUST do OpenClipboard before GetClipboardDataSize */
    if (! OpenClipboard())
        fail("Can't open clipboard -- try again later");
    clip_open = 1;

    if ((len = GetClipboardDataSize(CF_TEXT)) == 0)
        fail("No text in clipboard");

    /* printf("%lu bytes of text in clipboard\n", len); */

    /* the following can be removed in 32-bit code */
    if (len > 0xFFF0UL)
        fail("Sorry, can't handle more than 64k of text");

    if ((buf = _fmalloc(len+1)) == 0) /* add 1 for NULL terminate */
        fail("Insufficient memory");

    if (! GetClipboardData(CF_TEXT, buf))
        fail("Couldn't get clipboard text");

    /* finally, we've got our text */
    CloseClipboard();       /* close it BEFORE displaying string */
    buf[len] = '\0';        /* add NULL terminate */
    printf("%Fs\n", buf);   /* print FAR string */
    return 0;
}
```

Figure 6: **TESTCLIP.C is a DOS program that retrieves text from the Windows Clipboard.**

Break won't be processed *until* the I/O operation that prints the string. So, if the user closes up before `printf`, he's in relatively good shape.

These measures are not fully adequate, however, and they shouldn't have to be thought about anyway. The Clipboard-access library should take care of such details instead of forcing applications such as TESTCLIP to worry about them.

A second and bigger problem TESTCLIP reveals is that our Clipboard-access library is too low-level. All we really want are functions that make it possible to put text onto and get text out of the Windows Clipboard. These functions are described in Figure 7. Once we implement these higher-level functions we should be able to totally forget about the low-level details of opening the Clipboard, making sure it gets closed, getting its size if we're retrieving text, and so on. (Note that the functions could be extended to the non-text bitmap format that the WINOLDAP Clipboard API supports, of course.)

The resulting higher-level functions are shown in Figure 8. The function GetClipString() is really just our TESTCLIP program, now turned into a function. Note that GetClipString() takes care of allocating memory for the retrieved text. This memory can be freed with the (utterly simple) FreeClipString() function.

The discussion so far has centered on retrieving the Clipboard contents. The function PutClipStrLen() ties together everything needed to *change* the Windows Clipboard from a DOS program. Note that the Clipboard must be emptied and compacted before changing it.

Most important of all, the routines in Figure 8 include clip_init(), which installs an `atexit()` handler and a Ctrl-C handler. I used a C static variable, winoldap, to ensure that clip_init() gets called once and only once. The `atexit()` handler, called exit_func(), and the Ctrl-C handler, called sigint_handler(), simply call CloseClipboard() if clip_open is still TRUE. There's nothing terrifically exciting about this, but putting it all inside WINCLIP.C means that we never have to think about it again.

HIGH-LEVEL ACCESS FUNCTIONS
Partial Listing

```
/* is the clipboard available? */
int WindowsClipboard(void);

/* put "len" bytes of text on clipboard */
int PutClipStrLen(char *str, unsigned len);

/* put NULL-terminated text on clipboard */
int PutClipString(char *str);

/* get text from clipboard */
char *GetClipString(void);

/* free the retrieved text when done */
void FreeClipString(char *str);
```

Figure 7: **These are some of the desirable higher-level Clipboard-access functions.**

SOME SIMPLE UTILITIES With our higher-level Clipboard-access library, it now takes only a few lines of code to add some useful DOS/Windows hybrid utilities. (Such hybrids are DOS programs that require Windows to run.)

Figure 9 shows the program GETCLIP.C. It is similar to the earlier TESTCLIP program, but it uses the higher-level functions. In fact, at around 30 lines of code, it's really just a wrapper around the GetClipString() function.

PUTCLIP.C (shown in Figure 10) is more interesting, not only because changing things is intrinsically more interesting than just looking at them, but because there are a number of possible places from which PUTCLIP can get its input. Running

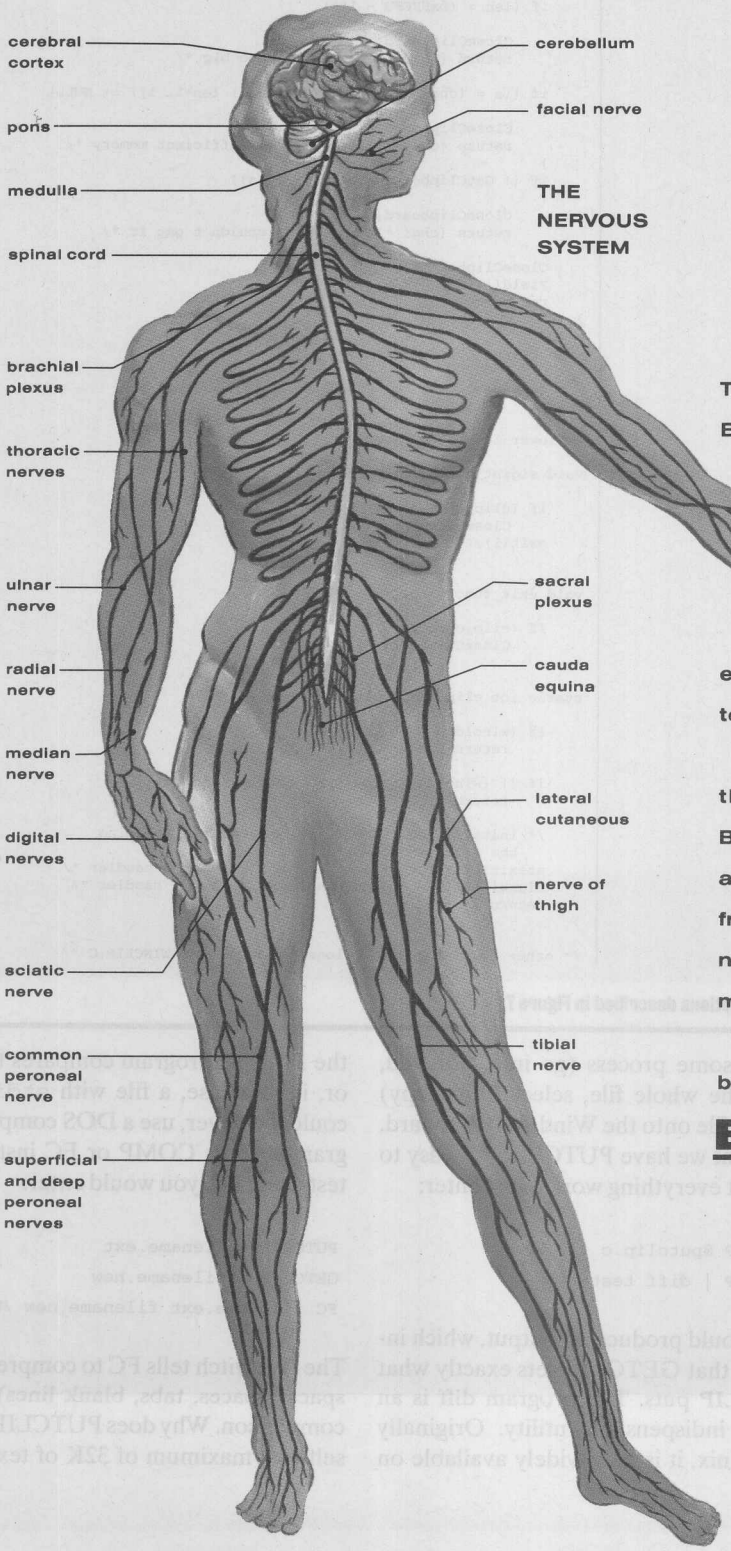
```
PUTCLIP "Hello World"
```

will, it should come as no surprise, put "Hello World" on the Windows Clipboard. The command

```
PUTCLIP @filename.ext
```

puts an entire file (up to 32K in length) on the Clipboard. Entering

A BAD DISK CAN DO SERIOUS DAMAGE TO YOUR SYSTEM.



There are two types of computer disks. Bad disks. And good disks.

And here's the problem: bad disks don't tell you they're bad until it's far too late. Suddenly your computer says you're experiencing a "fatal disk error." (Your body has a different term for it. Like panic.)

Fortunately, good disks can tell you they're good. They will simply say BASF right on the package. And they come from the people who not only invented magnetic recording, they perfected it.

The next time you're buying disks, be safe. BASF.

BE SAFE. BASF.

BASF

WINCLIP.C
Partial Listing

```
/*
WINCLIP.C -- DOS access to Windows Clipboard (Enhanced mode)

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <signal.h>
#include <dos.h>
#include "winclip.h"

static int winoldap = 0;
static int clip_open = 0;

static int clip_init(void);

/* higher-level functions */

int WindowsClipboard(void)
{
    return (IdentifyWinOldApVersion() != 0);
}

/* NOTE: len should include NULL-termination byte,
   which is required */
int PutClipStrLen(char *s, unsigned len)
{
    if (! winoldap)
        if (! clip_init())
            return 0;

    if (! OpenClipboard())      /* does clip_open++ */
        return 0;
    if (! EmptyClipboard())
    {
        CloseClipboard();       /* does clip_open-- */
        return 0;               /* couldn't empty */
    }
    if (CompactClipboard(len) < len)
    {
        CloseClipboard();
        return 0;               /* couldn't compact */
    }
    if (! SetClipboardData(CF_TEXT, s, len))
    {
        CloseClipboard();
        return 0;               /* couldn't set */
    }
    CloseClipboard();
    Yield();
    return 1;
}

int PutClipString(char *str)
{
    return PutClipStrLen(str, strlen(str)+1);
}

char *GetClipString(void)
{
    unsigned long len;
    char *s;

    if (! winoldap)
        if (! clip_init())
            return (char *) 0;

    if (! OpenClipboard())
        return (char *) 0;
    /* MUST do OpenClipboard BEFORE GetClipboardDataSize */
    if ((len = GetClipboardDataSize(CF_TEXT)) == 0)
    {
        CloseClipboard();
        return (char *) 0;      /* nothing there */
    }
    if (len > (0xFFFFU - 16))
    {
        CloseClipboard();
        return (char *) 0;      /* too big */
    }
    if ((s = (char *) calloc((unsigned) len+1, 1)) == NULL)
    {
        CloseClipboard();
        return (char *) 0;      /* insufficient memory */
    }
    if (! GetClipboardData(CF_TEXT, s))
    {
        CloseClipboard();
        return (char *) 0;      /* couldn't get it */
    }
    CloseClipboard();
    Yield();
    return s;
}

void FreeClipString(char *s)
{
    free(s);
}

/* lower-level functions */ .

void sigint_handler(int sig)
{
    if (clip_open != 0)
        CloseClipboard();
    exit(1);
}

void exit_func(void)
{
    if (clip_open != 0)
        CloseClipboard();
}

static int clip_init(void)
{
    if (winoldap)
        return 1;   /* already did init */

    if (! (winoldap = IdentifyWinOldApVersion()))
        return 0;

    /* install handlers so that we never exit holding
       the clipboard open */
    atexit(exit_func);              /* at-exit handler */
    signal(SIGINT, sigint_handler); /* Ctrl-C handler */
    return 1;
}

/* other lower-level functions are here; see WINCLIP.C */
```

Figure 8: **This section of WINCLIP.C implements the higher-level functions described in Figure 7.**

```
PUTCLIP -
```

puts its standard input on the Clipboard so that, for example, the command

```
DIR | PUTCLIP -
```

puts a DOS directory listing on the Windows Clipboard. Apart from handling these variants, though, PUTCLIP is just packaging for the function PutClip*StrLen()*.

Earlier, when checking that TESTCLIP worked, it required a somewhat cumbersome process (go into Notepad, select the whole file, select Edit Copy) to get a file onto the Windows Clipboard. Now that we have PUTCLIP, it's easy to test that everything works. Just enter:

```
PUTCLIP @putclip.c
GETCLIP | diff test.txt -
```

This should produce no output, which indicates that GETCLIP gets exactly what *PUTCLIP* puts. The program diff is an almost indispensable utility. Originally from Unix, it is now widely available on the PC. The program compares two files or, in this case, a file with `stdin`. You could, however, use a DOS compare program such as COMP or FC instead. To test using FC, you would enter:

```
PUTCLIP @filename.ext
GETCLIP > filename.new
FC filename.ext filename.new /W
```

The /W switch tells FC to compress wh space (spaces, tabs, blank lines) for the comparison. Why does PUTCLIP limit itself to a maximum of 32K of text? Since

GETCLIP.C

Complete Listing

```
/*
GETCLIP.C -- DOS program gets text from Windows clipboard
requires WINCLIP.C
for example (Borland C++): bcc getclip.c winclip.c

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "winclip.h"

void fail(char *s) { fputs(s, stderr); fputs("\n", stderr); exit(1); }

int main(int argc, char *argv[])
{
    char *s;

    fputs("GETCLIP version 1.0\n", stderr);
    fputs("Copyright (c) 1992 Ziff Davis Communications "
          "* Andrew Schulman\n\n", stderr);

    if ((argc > 1) && (argv[1][0]=='/') && (argv[1][1]=='?')) // /?
        fail("GETCLIP retrieves text from the Windows clipboard");

    if (! WindowsClipboard())
        fail("This program must run in a DOS box "
             "under Enhanced mode Windows");

    if (s = GetClipString())
    {
        puts(s);
        FreeClipString(s);
    }
    else
        puts("* clipboard empty *");
    return 0;
}
```

Figure 9: **GETCLIP.C is a DOS utility that retrieves text from the Windows Clipboard.**

PUTCLIP.C

Complete Listing

```
/*
PUTCLIP.C -- put text onto Windows clipboard
requires WINCLIP.C
for example (Borland C++): bcc putclip.c winclip.c

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>
#include "winclip.h"

void fail(char *s) { puts(s); exit(1); }

int main(int argc, char *argv[])
{
    unsigned len;
    char *p;

    fputs("PUTCLIP version 1.0\n", stderr);
    fputs("Copyright (c) 1992 Ziff Davis Communications "
          "* Andrew Schulman\n\n", stderr);

    if ((argc < 2) ||
        ((argc > 1) && (argv[1][0]=='/') && (argv[1][1]=='?'))) // /?
        fail(
    "PUTCLIP puts text onto the Windows clipboard\n"
    "usage: PUTCLIP text          puts text into clipboard\n"
    "       PUTCLIP @filename     puts file contents into clipboard\n"
    "       PUTCLIP -             copies standard input into clipboard\n");

    if (! WindowsClipboard())
        fail("This program must run in a DOS box "
             "under Enhanced mode Windows");

    if (! (argv[1][0] == '-' || argv[1][0] == '@'))
    {
        /* Put string from command line onto clipboard */
        static char buf[128];
        char far *cmdline = MK_FP(_psp, 0x82);
        int len = *((unsigned char far *) MK_FP(_psp, 0x80)) - 1;
        _fmemcpy(buf, cmdline, len);
        p=buf;
    }
    else
    {
        int f;
        unsigned rc;    /* count of bytes read */
        if (argv[1][0] == '-' && argv[1][1] == '\0')  // - on cmdline
            f = 0;      // STDIN
        else if (argv[1][0] == '@')   // filename on cmdline
            if ((f = open(&argv[1][1], O_RDONLY)) == -1)
                fail("can't open file");
        if ((len = filelength(f)) > (32 * 1024))// 32k max
            fail("file too big");
        if ((p = malloc(len+1)) == 0)
            fail("insufficient memory");
        if ((rc = read(f, p, len)) < 1) // means 32k max!
            fail("can't read file");
        close(f);
        p[rc] = '\0';   /* must be NULL terminated */
        len = rc;
    }

    if (PutClipStrLen(p, len+1))   // +1 for NULL
        puts("putclip successful");
    else
        puts("putclip failed");

    return 0;
}
```

Figure 10: **PUTCLIP.C is a DOS utility that puts text into the Windows Clipboard.**

the GetClipboardDataSize() function expects a 4-byte number, you would think we could move megabytes at a time onto the Clipboard. In fact, a 32-bit 386|DOS-Extender version of PUTCLIP was able to do just that, and the 32-bit version of GETCLIP retrieved them without a hitch. But I found that Windows utilities such as the Clipboard Viewer got hopessly confused by such large items.

Why do *PUTCLIP* and *GETCLIP* limit themselves to text? After all, the WINOLDAP Clipboard API also supports bitmaps. This, in fact, is probably why the API provides the seemingly out-of-place GetDeviceCaps() function—so that DOS programs can properly display bitmaps retrieved from the Clipboard. You may want to try to modify PUTCLIP and GETCLIP to handle bitmaps.

However, the purpose of our work so far has been to prepare for sending *requests* to Windows. These requests (or commands) will look just like text, but will be interpreted by a Windows program waiting patiently for them to appear in the Clipboard. How to write such a Windows program, and then how to write DOS programs that send the requests, will be discussed in Parts 2 and 3. When we're done, we'll have knocked down some of the walls around the DOS box. □

ANDREW SCHULMAN IS A WRITER AND ENGINEER AT PHAR LAP SOFTWARE IN CAMBRIDGE, MASSACHUSETTS. HE IS COAUTHOR OF THE BOOK *UNDOCUMENTED DOS* AND OF *UNDOCUMENTED WINDOWS* (FORTHCOMING), FROM ADDISON-WESLEY.

Buffered Input and Output Of MIDI Short Messages

BY CHARLES PETZOLD

As you'll recall, in the previous issue I complained about an apparent gap I perceived in the application programming interface (API) of Microsoft Windows with Multimedia Extensions: a gap in the support of the Musical Instrument Digital Interface (MIDI). I felt that the low-level API should have included functions for buffered input and output of MIDI short messages.

MIDI short messages are 1, 2, or 3 bytes in length. A MIDI controller (such as a keyboard) generates MIDI short messages when you press and release keys or when you manipulate dials or buttons on the keyboard. A MIDI synthesizer responds to the MIDI short messages by sounding tones. It would be nice to write a program that stores messages coming from a controller through the MIDI In port of a MIDI board and later plays them back through an internal synthesizer or an external one connected to the MIDI Out port of the board.

Currently you can use the low-level API to send and receive MIDI short messages only one at a time, either in a window procedure or a DLL (dynamic link library). It's most convenient to use a window procedure, of course, but then you don't get time stamps for MIDI input and you're forced to use the imprecise Windows timer for MIDI output.

For the degree of accuracy that music requires, you need to use a DLL. A callback function in the DLL can obtain MIDI messages from a keyboard or other controller with time stamps accurate to a millisecond. With a DLL, you can also use the high-precision multimedia timer for sending MIDI messages to a synthesizer, also with millisecond accuracy.

That you need to use a DLL to work with the low-level MIDI API really bothers me. First, there is the nuisance of separating program code into an executable and a DLL. Furthermore, the DLL must include functions that are called by the system during hardware interrupts, and these functions can be tricky to write.

EXTENDING THE API It would be most convenient to send and receive MIDI short messages using a buffer. For example, you could submit a buffer to the API to get MIDI input messages from a controller. This buffer, which would contain both the MIDI messages and timing information, could be returned to the program when full. You could then submit the buffer to the API to play the messages on a MIDI synthesizer. Because the API would handle all the timing, you wouldn't need to use a DLL. You could do everything from a window procedure.

Windows wouldn't be Windows if you couldn't extend the API. So I decided to define several new functions for buffered MIDI input and output.

Of course, Windows wouldn't be Windows if there weren't a way to extend the API. So I decided to correct the deficiency in the MIDI API by defining several new functions for buffered MIDI input and output. I had to use a dynamic link library for this, of course, but it's a DLL that can be used by several different types of MIDI programs. The DLL is responsible for implementing the callback functions that obtain MIDI input messages and use the multimedia timer for sending MIDI output messages.

Fortunately, some models for the new functions I had to invent already existed. The waveform audio API in Multimedia Windows uses buffers for both input and output, and even the MIDI API includes some functions for buffered input and output—albeit useful only for MIDI long (system-exclusive) messages.

I set a couple goals for myself: The new functions should look and work very much like existing API function calls, and the DLL should be written so that programmers could integrate the new functions into an existing program without having to make a slew of changes. Moreover, these new functions should not interfere with the existing API.

A NEW MIDI INPUT FUNCTION I began by defining the functions I thought should be added to the low-level MIDI API. The first was:

```
midiInShortBuffer (hMidiIn,
 lpMidiHdr, sizeof (MIDIHDR))
```

The word *short* in this function name does not refer to the length of the buffer, but indicates that the function is used for submitting a buffer to receive MIDI short messages. The function has the same syntax as midiInAddBuffer, which is used for submitting a buffer to obtain the long system-exclusive messages.

The first parameter is a handle to a MIDI input device; the second parameter is a pointer to a MIDIHDR structure. In this MIDIHDR structure, the lpData field must be set to a pointer to a buffer, and the dwBufferLength field must be set

to the size of this buffer. The buffer must be at least 8 bytes long, and preferably the size should be a multiple of 8 bytes.

As usual, both the MIDIHDR structure and the buffer must be allocated from global memory using the GMEM_MOVEABLE and GMEM_SHARE flags. Before calling midiInShortBuffer, the MIDIHDR structure must be passed to the midiInPrepareHeader function. This locks the structure and buffer in memory and prevents Windows from swapping them to disk.

Following a call to midiInShortBuffer, all short messages coming through the MIDI In port are accumulated in the buffer. (The single exception is the Active Sensing message that many MIDI controllers send out to indicate they are still connected.) The buffer is returned to the application when the buffer is full or when midiInReset is called. A buffer is returned to an application using the MM_MIM_LONGDATA message to a window procedure or an MIM_LONGDATA message to a callback function. These are the same messages used to return buffers submitted using the midiInAddBuffer call. (If a program is also using midiInAddBuffer to receive system-exclusive messages, it is the responsibility of the application to determine which is which.)

When the buffer is returned to the application, the dwBytesRecorded field of the MIDIHDR structure indicates how much data is in the buffer. This will always be a multiple of 8 bytes. The data in the buffer consists of a series of 32-bit unsigned integers, alternating between delta times (in milliseconds) and MIDI short messages packed into 32-bit integers. The first delta time is measured from the time midiInStart was called. Subsequent delta times indicate the millisecond delays between successive MIDI messages.

You can submit multiple buffers using midiInShortBuffer before the first buffer is returned. In fact, this is recommended so that you don't lose any messages between the time you get back one buffer and the time you submit another. The buffers can be larger than 64K.

NEW MIDI OUTPUT FUNCTIONS My next task was to define a similar function for MIDI output:

```
midiOutShortBuffer (hMidiOut,
 lpMidiHdr, sizeof (MIDIHDR))
```

This function works the same way as midiOutLongMsg. The format of the buffer is as described above, with alternating 32-bit delta times and 32-bit packed MIDI messages. The function plays the MIDI messages in the background until the buffer is finished or until midiOutReset is called. The function indicates that the buffer is finished by sending an MM_MOM_DONE message to a window procedure or making an MOM_DONE call to a callback function

You won't lose any messages between the time you get back one buffer and the time you submit another if you use midiInShortBuffer to submit multiple buffers.

in a DLL. This is the same message used for indicating that buffers submitted with midiOutLongMsg are finished. Multiple buffers can be submitted before the first buffer has been returned, and the buffers can be larger than 64K.

At this point, two additional functions become feasible, one to pause MIDI output

```
midiOutPause (hMidiOut)
```

and another to restart it:

```
midiOutRestart (hMidiOut)
```

These functions were inspired by waveOutPause and waveOutRestart, which are used for waveform audio output.

SMOOTH INTEGRATION As I thought about implementing these new functions in a DLL, it became obvious that the DLL would also have to include enhanced versions of some of the existing MIDI function calls. Some functions (such as midiOutPrepareHeader) didn't need to be tampered with. Others required additional functionality. For example, midiOutReset has to mark all pending short-message buffers as done and return them to the application. So midiOutReset had to be made to do what it does normally, and then perform additional work as well.

Of course, this new midiOutReset function must be able to identify the buffers associated with the handle to the MIDI output device. If only one program were allowed to use the DLL, this information could be stored in static variables in the DLL. But that didn't seem good enough. It would be preferable to allow two or more programs to use the DLL simultaneously with different MIDI ports.

This implied that all the information needed to implement the four new function calls had to be stored in the DLL's local heap space, probably as a small structure. A pointer to this structure would have to be identified by the handle passed as a first parameter to the MIDI functions. Thus, I couldn't use the real MIDI input or output handle in these functions. The handle had to be a pointer into the DLL's local heap. One field of the structure stored there would be the real handle to the MIDI device.

This meant that my DLL would have to include new versions of all the MIDI input and output functions that referred to the handle to the MIDI device. The new midiInOpen and midiOutOpen functions would call the existing open functions to obtain a MIDI input and output handle, allocate some memory in the DLL local heap for a data structure, store the real handle there, perform some initialization, and then pass back to the application the pointer to the DLL heap. The new midiInClose and midiOutClose functions would call the existing close functions and also free the local memory.

To the application, nothing would appear different. The application would still call the same MIDI functions, but they would be intercepted by the new DLL.

FUNCTION NAMES AND DLLs Of course, there's a little problem here. How do you define a function named midiInOpen in a new DLL when this function must call a function named midiInOpen in another DLL? It's not really as difficult as it may seem at first, just a little confusing.

Before I dive in, some background:

When you write a Windows program, you make calls to Windows functions such as midiInOpen. Of course, what you're implicitly doing is calling a particular function in a particular DLL, in this case MMSYSTEM.DLL. This crucial information is inserted into the program's .EXE file during linking. For a Windows program using multimedia function calls, you must link with the MMSYSTEM.LIB import library. This import library indicates that a call to a function named midiInOpen is really a call to a function in MMSYSTEM.DLL named midiInOpen with an ordinal number of 304.

The object here is to create a new DLL (let's call it MIDBUF.DLL) that exports functions named midiInOpen, and so forth. This DLL makes calls to midiInOpen and other functions in MMSYSTEM.DLL. An import library named MIDBUF.LIB could also be created. This import library would indicate that the function named midiInOpen is actually a call to MIDBUF.DLL. If you write an application and want to use the MIDBUF dynamic link library, you link with both MIDBUF.LIB and MMSYSTEM.LIB. On the link command line, MIDBUF.LIB must appear before MMSYSTEM.LIB.

What happens is that the application's .EXE file is set up so that the midiInOpen call is actually a call to the midiInOpen function in MIDBUF.DLL. The midiInOpen function in MIDBUF.DLL then makes a call to the midiInOpen function in MMSYSTEM.DLL.

The mechanics are simpler than you may think. In the MIDBUF.C source code file, the functions you want to intercept are given different names; for example, xMidiInOpen. This function can then make a call to the real midiInOpen function without the compiler thinking you're making recursive calls.

In the module definition file, you define xMidiInOpen as an alias in the EXPORTS section of the module definition file:

```
EXPORTS
    midiInOpen = xMidiInOpen
```

The function actually exported from the DLL is called midiInOpen, but this function is the same as xMidiInOpen in the C source code file. When you create the MIDBUF.LIB import library from the module definition file, the xMidiInOpen alias is ignored and the function appears in the import library as midiInOpen.

AND NOW THE IMPLEMENTATION The source code for the MIDBUF DLL grew to be a little larger than I expected, so it won't be printed here. The files, which include MIDBUF.MSC and MIDBUF.BCP (Make files for Microsoft C/C++ 7.0 and Borland C++ 3.0, respectively), MIDBUF.C, MIDBUF.DEF, and the dynamic link library MIDBUF.DLL, can be downloaded from PC MagNet and are archived as MIDBUF.ZIP. If you want to use the DLL in an application, you'll need two other files, also included in MIDBUF.ZIP: MIDBUF.H, a header file that defines the four new functions, and the import library MIDBUF.LIB.

The new midiInOpen function in MIDBUF first allocates memory in the DLL's local heap for a small data structure. Normally, when you open a MIDI input device, you specify in the midiInOpen function whether you want to be notified of MIDI messages through a window procedure or a callback function in a DLL. You can also specify application-defined data that is passed to the callback function.

Specify in midiInOpen whether you want to be notified of MIDI messages through a window procedure or a callback function.

The new midiInOpen function stores this information in the data structure that was just allocated. The new function then calls the old midiInOpen function, always specifying a callback function in MIDBUF.DLL. The application-defined data field in the open call is set to the data structure pointer. If the midiInOpen call is successful, the input and output handle is also stored in the data structure, and the pointer to the data structure is passed back to the application.

In subsequent MIDI input calls, the handle is passed as the first parameter to the function. The functions in MIDBUF.DLL use this handle as a pointer into the local heap, and thus can easily retrieve the real MIDI input handle. The callback function also has access to this data structure, and can then post a message to the application's window procedure or call another callback function, depending on what the application requested.

When buffers are passed to the new midiInShortBuffer function, they are maintained as a linked list. If you take a look at the definition of the MIDIHDR structure in the *Microsoft Windows Multimedia Programmer's Reference,* you'll see a field called lpNext, which is defined as a far pointer to a MIDIHDR structure. The documentation cautions that this structure field is "reserved and should not be used," but it's obvious that its purpose is for linking buffers.

In the MIDI input callback function in MIDBUF.DLL, incoming MIDI messages are indicated by an MIM_DATA call. These MIDI messages are stored in the current buffer. When the buffer becomes full, the callback function notifies the application through the use of an MM_MIM_LONGDATA message posted to a window procedure, or through a MIM_LONGDATA call to the application-requested callback function.

For buffered MIDI output, MIDBUF.DLL needs to use the multimedia timer routines to set a high-precision timer. This timer also requires a callback function in the DLL. For the first buffer submitted to midiOutShortBuffer, timeSetEvent is called using the first delta time in the buffer. During the timer callback function, the MIDI message is sent to the device using midiOutShortMessage. The timeSetEvent function is then called again with the next delta time. When the buffer is finished, the application is notified through the use of an MM_MOM_DONE message posted to a window procedure or an MOM_DONE call to the application-requested callback function.

LET'S TRY IT OUT In the next issue, I'll present a program called MIDREC that uses MIDBUF to record MIDI messages coming from a keyboard or other MIDI controller and then plays them through a MIDI synthesizer. □

LAB NOTES

Accessing the Windows API From the DOS Box, Part 2

BY ANDREW SCHULMAN

If you've ever had unwelcome guests attend a party, you may have tried to get rid of them by "forgetting" to refill their drinks or neglecting to introduce them to other guests; actually, doing everything short of kicking them out: "Here's your hat. What's your hurry?"

This is how Microsoft Windows treats DOS programs. It provides these unwelcome guests with a level of service that is just short of insulting. This creates an unfortunate situation for those of us who like Windows but who also have many DOS programs that we need to run. It would be nice if somehow our DOS programs could benefit from running under Windows.

In last issue's Lab Notes we saw that, although Windows generally treats DOS programs like unbidden guests, it does provide at least a few programmer's services that we can use to build a bridge between DOS programs and Windows functionality. More specifically, we saw that the INT 2Fh functions of Windows Enhanced mode provide a way for DOS applications to access the Windows Clipboard.

We used these INT 2Fh functions—OpenClipboard(), GetClipboardData(), SetClipboardData(), CloseClipboard(), and so on—to build two C functions: PutClipString() and GetClipString(). In turn, we used these two functions to build two handy DOS programs—GETCLIP and PUTCLIP—which retrieve or change the contents of the Windows Clipboard.

But as nice as it is to provide DOS programs with a C function that lets them read the Windows Clipboard, we surely want to do more than that. Moreover, if Microsoft were determined to give DOS programs access to only 10 of the roughly 1,200 Windows API functions, why pick the Clipboard functions instead of more useful ones?

If you could call the WinExec() function from a DOS program, for example, you could run Windows programs from the DOS box. Calling SetWindowText() would allow you to change a window's title bar. And if you were able to call SendMessage() or PostMessage(), a DOS program could send keystrokes to a Windows application. Microsoft did not provide DOS programs with INT 2Fh versions of WinExec() or SetWindowText() or SendMessage() or PostMessage(), however. So, with nothing but OpenClipboard(), SetClipboardData(), and the like, are we stuck?

By learning to construct CLIPSERV.C and CMDCLIP.C, you'll be able to start giving your DOS programs access to the Windows API.

No. The INT 2Fh Clipboard functions *will* suffice. DOS programs can use this handful of functions to gain a limited form of access to the Windows party. True, there's no "Open Sesame" command that will admit DOS programs directly to the party—nothing will change that. But a DOS program running under Windows *can* use the Clipboard to communicate its requests to a responsive Windows application.

Thus, although you can't run a Windows application from the DOS box, you can ask a suitable Windows application to run one for you. You can't change the title bar, but again, you can ask a Windows application to do it on your behalf. What you need is a *surrogate program*: a Windows application whose sole responsibility is to do things for your DOS programs that they aren't allowed to do for themselves. I'll show you how to write such a program—a Windows application that acts as a *server* for requests that come from DOS applications—using the Clipboard as the place where these messages get delivered.

Before I proceed, however, it's worth mentioning that there are other possible approaches to the problem of giving DOS applications access to Windows. For example, Doug Boling's WINSTART utility (*PC Magazine,* June 30, 1992), uses a DOS TSR that runs *before* Windows to let you run Windows programs from within a DOS box.

Alternatively, Microsoft C/C++ 7.0 includes a Windows virtual device driver (VMB.386) that provides a message buffer that is shared between a DOS program (WX) and a Windows program (WXSRVR, the Windows Spawn Server). With this approach, however, you need three executables just to be able to type the name of a Windows program in the DOS box and have it do something other than print "This program requires Microsoft Windows"!

My approach differs from those of WINSTART and WXSRVR not only in that it uses the Clipboard as the message buffer, but also in that it provides DOS programs with general-purpose access to the Windows API, not just to WinExec(). I'm going to create two programs: CLIP-

SERV, the Windows server, and CMDCLIP, the DOS client. By the time we're done, CLIPSERV will provide CMDCLIP with what in Windows is called run-time dynamic linking. Also, the ability to call the WinExec() function from the DOS box will just fall out from this as one example of a general-purpose capability. Programs that were mentioned in Part 1 of this discussion (in the August 1992 issue) are available on PC MagNet, archived as CLIP1.ZIP, CLIPSERV, and CMDCLIP. The related files introduced in this second part can be downloaded as CLIP2.ZIP.

Let's start with the PUTCLIP program from last issue. PUTCLIP is a DOS program that puts text on the Windows Clipboard. For example,

```
C:\PCMAG>putclip "hello Windows!"
```

puts the string "hello Windows!" on the Windows Clipboard. You can see this string appear if you run the Windows Clipboard program (CLIPBRD.EXE), which displays the Clipboard's current contents.

Now lets put a different string in the Clipboard. Figure 1 shows the new string—RUN NOTEPAD.EXE—which looks like a request. But no matter how much that string *looks* like a request, there's no reason to believe that it will be *acted upon* as a request. As Glendower, in Shakespeare's *Henry IV, Part I,* declared: "I can call spirits from the vasty deep." To which Hotspur properly retorted: "Why, so can I, or so can any man; But will they come when you do call for them?"

In other words, the Clipboard Viewer (CLIPBRD.EXE) that comes with Windows has no way to treat a string as a command line. Fortunately, although the Viewer is frequently confused with the Clipboard itself, in reality the Viewer is just another application. Moreover, there is provision within Windows to replace it or supplement it with alternate Clipboard Viewers. In fact, instead of displaying the Clipboard's current contents, an alternative Viewer could actually *do* something, such as carry out the request that appears in the Clipboard. With such a Windows program hanging off the Clipboard, DOS programs can indeed call Windows spirits from the vasty deep!

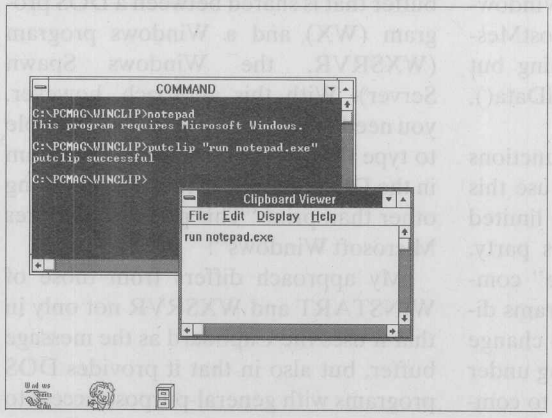


Figure 1: **PUTCLIP puts a string into the Clipboard, and CLIPBRD displays it.**

Instead of displaying the Clipboard's current contents, an alternative Viewer could actually do something—such as carry out the request that appears in the Clipboard.

HANGING OFF THE CLIPBOARD Our first goal, then, is to write a Windows program that waits for DOS programs to put requests into the Clipboard. This program, CLIPSERV, needs to know: first, when the Clipboard contents have changed; second, whether they were changed by a DOS program; and third, whether the DOS program is issuing a request rather than simply engaging in a normal cut-and-paste operation.

Taken individually, each of these problems is easy to solve. CLIPSERV can find out that the Clipboard contents have changed not by checking the Clipboard every second, but by doing exactly what Windows programs always do: waiting until a message arrives. Windows lets a program know that the Clipboard contents have changed by sending it a WM_DRAWCLIPBOARD message. The message gets its name from the fact that Windows expects the recipient to display the entire Clipboard contents; however, it really means only that there has been a change in the Clipboard contents.

In order to receive the WM_DRAWCLIPBOARD message, CLIPSERV must use the SetClipboardViewer() function to set itself up as a Clipboard Viewer. Since there can be more than one program hanging off of the Windows Clipboard at a given time, each program in the Clipboard Viewer chain is expected to help keep the chain intact. Therefore, alternative viewers must also handle WM_CHANGECBCHAIN messages and pass on every WM_DRAWCLIPBOARD message.

Inside the part of the program that handles the WM_DRAWCLIPBOARD messages, CLIPSERV can string together three Windows function calls to figure out whether the Clipboard contents were changed by a DOS program:

- GetClipboardOwner() returns the window handle (HWND) of the last program to call SetClipboardData().
- GetWindowTask() takes an HWND and returns a correponding task handle (HTASK); that is, an identifier for an instance of a program.
- IsWinOldApTask() takes an HTASK and determines whether the task belongs to the Windows WINOLDAP module; that is, whether it is a DOS ("old") application. This function is undocumented but reliable. Indeed, Windows' own TASKMAN uses it. (For more details about IsWinOldApTask(), see *Undocumented Windows,* a book I recently coauthored.)

So far, CLIPSERV knows whenever a program in the DOS box has put something in the Clipboard. Now CLIPSERV needs a way to distinguish between command strings and normal cut-and-paste text. This is the problem with hijacking the Clipboard Viewer as an interprocess-communications mechanism.

In Windows programming, the RegisterClipboardFormat() function would be used to enable such private conversations. In this case, however, there's a DOS program on one end of the conversation; and unfortunately, RegisterClipboardFormat() is not one of the Windows API calls that Microsoft supplied with an INT 2Fh interface. I mentioned earlier

that CLIPSERV would give DOS programs general-purpose access to the Windows API and, in fact, a DOS program *will* be able to call RegisterClipboardFormat()—as soon we can set up a reliable connection to CLIPSERV!

It's one of those chicken-and-egg–type problems. The solution, at least for the time being, is to have the DOS program put its requests in the Clipboard using a standard CF_TEXT format. Although CF_TEXT (the CF simply stands for Clipboard Format) is also used by WINOLDAP *whenever* you copy text from a DOS box to the Clipboard, we can establish a convention of our own that will identify the string as a request to be carried out rather than ordinary, static text.

Since the DOS client that we will eventually be building will be called CMDCLIP (CMD is just an abbreviation for command), I arbitrarily ruled that any text placed in the Clipboard by a DOS box that started with the string "CMDCLIP" was to be interpreted as a command to be handled by CLIPSERV. Call it a kludge if you like, but it works fine in practice.

Conceptually, the key part of CLIPSERV looks like the semi-pseudocode shown in Figure 2.

CLIPSERV PSEUDOCODE
Partial Listing

```
switch WM_DRAWCLIPBOARD:
    if (IsWinOldApTask(GetWindowTask(GetClipboardOwner())))
    {
        char *s = get_clipboard_string();
        if (strncmp(s, "CMDCLIP", 7) == 0)
            do_request(&s[8]);
    }
```

Figure 2: **In this conceptual overview of the key component of CLIPSERV, strings beginning with CMDCLIP are interpreted as commands.**

GETMSG.C
Complete Listing

```
/* GETMSG.C — GetMessage with no window */

#include ``windows.h"

int PASCAL WinMain(HANDLE hInst, HANDLE hPrevInst,
    LPSTR lpCmdLine, int nCmdShow)
{
    WORD hTimer;
    MSG msg;
    int i = 0;

    hTimer = SetTimer(0, 1, 1000, NULL);
    while (GetMessage(&msg, NULL, 0, 0))
        if (msg.message == WM_TIMER)
        {
            MessageBeep(0);
            if (++i == 5)
                break;
        }
    KillTimer(0, hTimer);
    return 0;
}
```

Figure 3: **This small Windows program illustrates windowless message handling.**

CLIPSERV
Partial Listing

```
WinMain()
{
    hwnd = objwnd(hPrevInstance, hInstance, "clipserv");
    hwndNextViewer = SetClipboardView(hwnd);
    on(WM_DRAWCLIPBOARD, do_checkclipboard);
    on(WM_CHANGECBCHAIN, do_changecbchain);
    on(WM_DESTROY, do_destroy);
    on(WM_CLIPCMD, do_clipcmd); // user-defined message
    return mainloop();          // go!
}
long do_checkclipboard(HWND hwnd, WORD msg, WORD wparam, LONG lparam)
{
    // heart of CLIPSERV: handle WM_DRAWCLIPBOARD here
}
long do_changecbchain(HWND hwnd, WORD msg, WORD wparam, LONG lparam)
{
    // handle WM_CHANGECBCHAIN here
}
```

Figure 4: **This code fragment shows the basic structure of CLIPSERV using the OBJWND.C library.**

HANDLING MESSAGES Like any Windows program, CLIPSERV must handle messages. As an ostensible Clipboard "viewer," CLIPSERV needs to be able to handle WM_DRAWCLIPBOARD, WM_CHANGECBCHAIN, and WM_DESTROY messages, at the very least. On the other hand, CLIPSERV has no need for a user interface: There's nothing for it to display in a window (unless you want to provide a debugging output), and it has no use for user input. Everything it needs to know it learns from the Clipboard.

This presents an odd problem. CLIPSERV is message-driven, yet it has no user interface. But in Windows programming, you never see message handling without a window. The function that handles messages is invariably called a WndProc, and the things that users perceive as windows are really just the user-interfaces for message handling. Yet there doesn't seem to be a good reason why the two concepts—message handling and user interface—should be linked in this way.

In fact, the message queues inside Windows *don't* have anything to do with windows. From a Windows programmer's perspective, you can call GetMessage() without having a window handle (HWND). All GetMessage() needs to work is a Task Queue structure, which is allocated automatically for your task even before its WinMain() function is called. What does require an HWND is DispatchMessage(), but this function is not essential to message handling.

In other words, it is possible to write windowless Windows programs that do background processing. A good example

is shown in Figure 3. The program GETMSG installs a timer that will go off once per second. It calls GetMessage(), yet it has no window. Each time it receives a WM_TIMER message, it beeps; after 5 seconds, it exits.

We could use this scheme in the CLIPSERV program, watching for WM_DRAWCLIPBOARD events instead of WM_TIMER events. But while not required for the basic message handling, CLIPSERV will need an HWND for other operations, such as Clipboard access.

What CLIPSERV really needs is something such as OS/2 Presentation Manager's *object window*. This is an entity that can be used as a target for messages and can handle messages, but is not visible and accepts no user input. Fortunately, it is possible to emulate OS/2 object windows under Microsoft Windows.

To handle the messages, of course, you must write what's called a *window procedure*. The term window here is really a misnomer, however: What is called a window procedure is really just a message handler. The window to which it corresponds can be invisible. It's not really a user-interface object but a pure message-handling object. The necessary RegisterClass() and CreateWindow() calls can be packaged into a function that creates something like the OS/2 object window.

At the same time, there are better ways to set up message handling than with the C switch/case statements usually used in Windows programs. These switch/case statements lead to single functions that go on for pages and pages, with no attempt to break the problem

OBJWND.C

Complete Listing

```
/* OBJWND.C -- "object windows" from WMHANDLER */

#include "windows.h"
#include "objwnd.h"

static WMHANDLER wmhandler[WM_USER] = {0};

#define MAX_EXTRA   32

typedef struct {
    WORD message;
    WMHANDLER handler;
    } EXTRAHANDLER;

static EXTRAHANDLER extrahandler[MAX_EXTRA] = {0};
static int num_extra = 0;

static int isextramsg(WORD message)
{
    EXTRAHANDLER *pex;
    int i;
    for (i=0, pex=extrahandler; i<MAX_EXTRA; i++, pex++)
        if (pex->message == message)
            return i;
    return -1;
}

static BOOL did_init = FALSE;
void objwnd_init(void)
{
    EXTRAHANDLER *pex;
    WMHANDLER *pwm;
    int i;
    for (i=0, pwm=wmhandler; i < WM_USER; i++, pwm++)
        *pwm = defwmhandler;
    for (i=0, pex=extrahandler; i < MAX_EXTRA; i++, pex++)
    {
        pex->message = 0;
        pex->handler = defwmhandler;
    }
    did_init = TRUE;
}

long FAR PASCAL _export WndProc(HWND hWnd, WORD message,
    WORD wParam, LONG lParam)
{
    int iExtraMsg;

    if (message < WM_USER)
        return (*wmhandler[message])(hWnd, message, wParam, lParam);
    else if ((iExtraMsg = isextramsg(message)) != -1)
        return (*extrahandler[iExtraMsg].handler)(hWnd,
            message, wParam, lParam);
    else
        return DefWindowProc(hWnd, message, wParam, lParam);
}

static long defwmhandler(HWND hwnd, WORD message, WORD wParam, LONG lParam)
{
    return DefWindowProc(hwnd, message, wParam, lParam);
}

WMHANDLER on(unsigned message, WMHANDLER f)
{
    WMHANDLER oldf;
    int iExtraMsg;

    if (! did_init)
        objwnd_init();

    if (message < WM_USER)
    {
        oldf = wmhandler[message];
        wmhandler[message] = f ? f : defwmhandler;
        return (oldf) ? oldf : defwmhandler;
    }
    else if ((iExtraMsg = isextramsg(message)) != -1)
    {
        oldf = extrahandler[iExtraMsg].handler;
        extrahandler[iExtraMsg].handler = f ? f : defwmhandler;
        return (oldf) ? oldf : defwmhandler;
    }
    else if (num_extra < MAX_EXTRA)
    {
        extrahandler[num_extra].message = message;
        extrahandler[num_extra].handler = f;
        num_extra++;
        return defwmhandler;
    }
    else
        return -1;  // couldn't set!
}

HWND objwnd(HANDLE hPrevInst, HANDLE hInst, char *name)
{
    static HWND hwnd = 0;

    if (hwnd)
        return hwnd;

    if (! did_init)
        objwnd_init();

    if (! hPrevInst)
    {
        WNDCLASS wndclass;
        memset(&wndclass, 0, sizeof(wndclass));
        wndclass.lpfnWndProc = WndProc;
        wndclass.hInstance = hInst;
        wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndclass.lpszClassName = "OBJECTWND";
        if (! RegisterClass(&wndclass))
            return 0;
    }

    hwnd = CreateWindow("OBJECTWND", name,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 0, 0,
        NULL, NULL, hInst, NULL);
    return hwnd;
}

void yield(void)
{
    MSG msg;
    if (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

int mainloop(void)
{
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

Figure 5: **OBJWND.C provides on-event message handling and creates the equivalent of the OS/2 Presentation Manager's invisible object windows.**

DefWindowProc() function that comes with the Windows SDK (Software Development Kit), for example, is 14 printed pages long! The natural tendency of switch/case statements is to produce such multipage functions.

Probably the best model for message-driven programming is the humble

```
ON event GOSUB subroutine
```

statement in BASIC. Interestingly, Microsoft has finally (after seven years!) acknowledged this rather obvious fact by introducing OnXXX "message crackers" in the Windows 3.1 SDK, and OnXXX handler functions in C/C++ 7.0. Alas, rewrite that 14-page DefWindowProc().

Fortunately, on-event programming is simple to implement for yourself; all it requires is breaking out of the old switch/case mindset of the SDK. For about two years I have been using on-event functions in every Windows program I write, thanks to a small set of functions called WMHANDLER that David Maxey of Lotus Development Corp. and I wrote together.

Seeing the need in CLIPSERV for essentially windowless message-handling (and having WMHANDLER as a base), I came up with a small C library called OBJWND. Using OBJWND, the basic structure of CLIPSERV looks like the

Instead of using a single, massive switch statement, CLIPSERV makes use of a collection of small functions, each of which handles a single message. To make this possible, OBJWND maintains an array of function pointers, called wmhandler[]. The array is indexed by Windows WM_message numbers such as 0x308 for WM_DRAWCLIPBOARD or 0x0002 for WM_DESTROY. For the most part, the on(msg, func) function just sets wmhandler[msg] = func. OBJWND has a built-in window procedure, WndProc(), whose sole job on receipt of a message is to call the function at wmhandler[message]. Whatever a given program does, this WndProc() function

CLIPSRV1.C

Complete Listing

```
/* CLIPSRV1.C -- first version of clipboard server */

#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include "windows.h"
#include "objwnd.h"

#define WM_CLIPCMD      (WM_USER+1)

HWND this_hwnd, hwndNextViewer;

void do_run(HWND hwnd, char far *cmd)
{
    WinExec(cmd, SW_NORMAL);
}

void do_settitle(HWND hwnd, char far *cmd)
{
    SendMessage(hwnd, WM_SETTEXT, 0, cmd);
}

#pragma argsused
long clipcmd(HWND hwnd, unsigned msg, WORD wparam, LONG lparam)
{
    HWND owner = wparam;    // HWND of clipboard owner
    char far *s = (char far *) lparam;
    char far *cmd = _fstrtok(s, " \t");
    char far *param = s + _fstrlen(cmd) + 1;
    if (_fstrcmp(cmd, "RUN") == 0)
        do_run(owner, param);
    else if (_fstrcmp(cmd, "SETTITLE") == 0)
        do_settitle(owner, param);
    else if (_fstrcmp(cmd, "EXIT") == 0)
        SendMessage(hwnd, WM_DESTROY, 0, 0L);
    _ffree((char far *) lparam);
    return 0;
}

long check_clipboard(HWND hwnd, unsigned msg, WORD wparam, LONG lparam)
{
    extern BOOL FAR PASCAL IsWinOldApTask(HANDLE); /* undocumented */
    char far *fp = 0;
    HWND owner;

    if (hwndNextViewer)
        SendMessage(hwndNextViewer, msg, wparam, lparam);

    /* only process CMDCLIP requests from OLDAPs */
    owner = GetClipboardOwner();
    if (IsWinOldApTask(GetWindowTask(owner)))
    {
        HANDLE hGMem;
        if (! OpenClipboard(hwnd))
            return 0;
        if (hGMem = GetClipboardData(CF_TEXT))
        {
            LPSTR lp = GlobalLock(hGMem);
            if (_fstrncmp(lp, "CMDCLIP ", 8) == 0)
            {
                int len = _fstrlen(lp);
                if ((lp[len-1] == '\n') || (lp[len-1] == '\r'))
                    lp[len-1] = '\0';   // remove CRLF
                if ((fp = _fmalloc(len+1)) != 0)
                {
                    char far *fp1 = fp;
                    char far *fp2 = &lp[8];     // remove "CMDCLIP "
                    while (*fp1++ = *fp2++)
                        ;
                }
                else
                    ;   /* insufficient memory */
            }
            GlobalUnlock(hGMem);
        }
        CloseClipboard();
    }
    // don't send message until clipboard closed
    if (fp)
        SendMessage(hwnd, WM_CLIPCMD, owner, fp);
    return 0;
}

long changecbchain(HWND hwnd, unsigned msg, WORD wparam, LONG lparam)
{
    /* maintain linked list of clipboard viewers */
    if (wparam == hwndNextViewer)
        hwndNextViewer = LOWORD(lparam);
    else if (hwndNextViewer)
        SendMessage(hwndNextViewer, msg, wparam, lparam);
    return 0;
}

#pragma argsused
long destroy(HWND hwnd, unsigned msg, WORD wparam, LONG lparam)
{
    ChangeClipboardChain(this_hwnd, hwndNextViewer);
    PostQuitMessage(0);
    return 0;
}

#pragma argsused
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    if (hPrevInstance)
    {
        MessageBox(NULL, "Already installed", "CLIPSERV", MB_OK);
        return 1;       // only one instance allowed
    }

    this_hwnd = objwnd(hPrevInstance, hInstance, "clipserv");
    hwndNextViewer = SetClipboardViewer(this_hwnd);

    on(WM_CHANGECBCHAIN, changecbchain);
    on(WM_DRAWCLIPBOARD, check_clipboard);
    on(WM_DESTROY, destroy);
    on(WM_CLIPCMD, clipcmd);    /* user-defined message */

    return mainloop();          /* go resident */
}
```

Figure 6: **CLIPSERV.C turns strings sent from a DOS program into Windows API calls.**

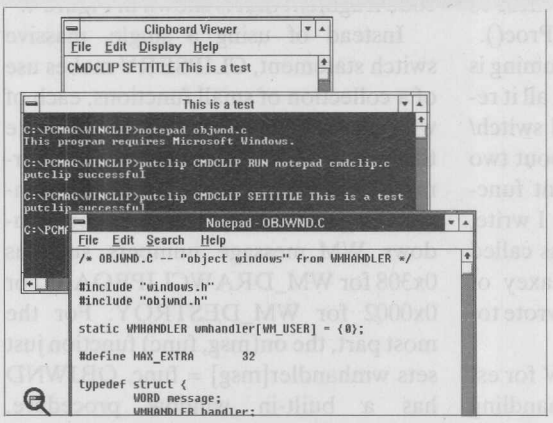


Figure 7: **In this screen shot, NOTEPAD is being run from the DOS box, with help from the (invisible) CLIPSRV1.**

never changes. The program's message-handling behavior is entirely set with calls to the on() function.

OBJWND.C is shown in Figure 5. You may be interested in comparing my method with that used by Ray Duncan in his Power Programming column (*PC Magazine,* May 26, 1992). Ray uses a static table-driven approach to message handling rather than the on() function call provided by OBJWND. The important point to note, however, is that neither of us uses a switch statement to handle messages. There is no reason why you should, either, unless you can hold the entire 14-page subroutine in your head.

Some of the object-oriented application frameworks for Windows, such as Borland's Object Windows Library (OWL), also get rid of the huge switch statement, replacing it with a collection of functions that each handle a single message. OBJWND.C shows one way such a scheme can be implemented. It's also worth noting that getting rid of the switch statement does not require an object-oriented language or an application framework, just a bit of common sense.

TURNING STRINGS INTO COMMANDS

With OBJWND.C in hand, we can proceed to write our alternative Clipboard server, CLIPSERV, which will turn a string like

```
CMDCLIP WINEXEC notepad 1
```

into the live request

```
WinExec("notepad", 1)
```

Before tackling this full-blown version, however, let's experiment a bit with a limited version so that we aren't overwhelmed and distracted by the additional code to handle runtime dynamic linking.

As shown in Figure 6, CLIPSRV1.C handles only three different requests: RUN, SETTITLE, and EXIT. As the code shows, each of these requests turns into a simple Windows API call that CLIPSRV1 is ready to carry out at a DOS program's behest:

- If CLIPSRV1 is running and a DOS program such as PUTCLIP puts the string

```
CMDCLIP RUN notepad.exe
```

into the Clipboard, it will effectively be as if the DOS program had called

```
WinExec("notepad.exe", SW_NORMAL);
```

- Likewise, the string

```
CMDCLIP SETTITLE This is a test
```

turns into

```
SendMessage(self, WM_SETTEXT, 0,
 "This is a test")
```

where *self* is the DOS box's own window handle.

- Finally, CMDCLIP EXIT causes CLIPSRV1 to send itself a WM_DESTROY message. Since CLIPSRV1 has no user interface, we will need a way to remove the program from memory, or CLIPSRV1 would act like a DOS TSR without an uninstall switch!

CLIPSRV1 does not care how you get any of these strings into the Clipboard, as long they come from a DOS box. The

DO_RUN() AND PUT_CLIP_STR()

Complete Listing

```
void do_run(HWND hwnd, char far *cmd)
{
    char buf[128];
    wsprintf(buf, "CLIPREPLY RUN %04X", WinExec(cmd, SW_NORMAL));
    put_clip_str(hwnd, buf);
}

WORD put_clip_str(HWND hwnd, char far *s)
{
    WORD ret;
    HANDLE h;
    if (! (h = GlobalAlloc(GMEM_MOVEABLE, _fstrlen(s)+1)))
        return 0;  /* insufficient memory */
    while (! OpenClipboard(hwnd))
        yield();   /* see OBJWND.C */
    EmptyClipboard();
    _fstrcpy(GlobalLock(h), s);
    GlobalUnlock(h);
    if (! (ret = SetClipboardData(CF_TEXT, h)))
        GlobalFree(h);
    CloseClipboard();
    return ret;
}
```

Figure 8: **The modified do_run() and additional put_clip_str() functions allow the CLIPSERV RUN command to put the WinExec return value back in the Clipboard.**

GETCLIP.BAT

Complete Listing

```
C:\PCMAG>type cliprun.bat
@echo off
putclip CMDCLIP RUN %1 %2 %3 %4 %5 %6 %7 %8 %9 > nul
getclip

C:\PCMAG>cliprun notepad
CLIPREPLY RUN 121E

C:\PCMAG>cliprun foobar
CLIPREPLY RUN 0002
```

Figure 9: **GETCLIP.BAT uses GETCLIP and PUTCLIP in a batch file to get replies to to CMDCLIP requests.**

CMDCLIP.C

Complete Listing

```
/*
CMDCLIP.C - put commands on Windows clipboard, for CLIPSERV
requires WINCLIP.C
for example (Borland C++): bcc cmdclip.c winclip.c

Copyright (c) 1992 Ziff Davis Communications
PC Magazine * Andrew Schulman (June 1992)
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include "winclip.h"

/* Save away contents of clipboard before changing it */
static char *save = 0;

/* Before exiting, make sure we restore the saved clip contents */
void cleanup(char *s, int ret)
{
    if (s)
    {
        fputs(s, stderr);
        fputc('\n', stderr);
    }
    if (save)
    {
        PutClipString(save);    /* put it back on clipboard */
        FreeClipString(save);
    }
    exit(ret);
}

int Clipserv(void)
{
    char *s;
    int i;
    PutClipString(``CMDCLIP INSTCHECK'');
    for (i=0; i<5; i++)
    {
        Yield();
        s = GetClipString();
        if (strcmp(s, "CLIPREPLY INSTOK'') == 0)
        {
            FreeClipString(s);
            return 1;
        }
        else
            FreeClipString(s);
    }
    /* still here */
    return 0;
}

int main(int argc, char *argv[])
{
    char buf[256], *s;
    char far *cmdtail;
    int len;
    int i;

    fputs(``CMDCLIP version 1.0\n", stderr);
    fputs(``Copyright (c) 1992 Ziff Davis Communications "
          "* Andrew Schulman\n\n", stderr);

    if ((argc < 2) ||
        ((argc > 1) && (argv[1][0]=='/') && (argv[1][1]=='?'))) // /?
        cleanup(
        "CMDCLIP puts commands onto the Windows clipboard, to be\n"
        "carried out by CLIPSERV\n\n"
        "usage: CMDCLIP clipboard-command [optional parameters. . .]\n"
        "clipboard commands:\n"
        "   DYNLINK [function]
[args. . .] - call a Windows API function\n"
        "   EXIT - tell CLIPSERV to uninstall itself\n"
        "   RUN [program] [args. . .] - run a Windows program\n"
        "   SETTITLE [string] - set the DOS box's title bar", 1);

    /* Make sure we're running under Windows */
    if (! WindowsClipboard())
        cleanup(``This program requires Windows Enhanced mode", 1);

    /* Save away current contents of clipboard, except if
       it's just an old CMDCLIP request.  NOTE:  CMDCLIP just
       saves/restores CF_TEXT; graphics will be bashed.
    save = GetClipString();
    if (strncmp(save, "CMDCLIP ", 8) == 0)
    {
        FreeClipString(save);
        save = 0;
    }

    /* Make sure that CLIPSERV is running:  Clipserv() install check
       works by putting a request into the clipboard and seeing if it
       gets changed in the right away.  If it doesn't, CLIPSERV must
       not be running.  But this test bashes the clipboard, so it
       should only be done AFTER we've saved the clipboard's contents */
    if (! Clipserv())
        cleanup(``This program requires Clipserv", 1);

    /* Assemble the command/request for CLIPSERV */
    strcpy(buf, "CMDCLIP ");
    cmdtail = MK_FP(_psp, 0x82);
    len = *((unsigned char far *) MK_FP(_psp, 0x80)) - 1;
    _fstrncat(buf, cmdtail, len);

    /* Send the request to CLIPSERV */
    PutClipString(buf);

    /* if asked server to exit, not going to be a reply! */
    if (strcmp(argv[1], "EXIT'') != 0)
    {
        /* Wait for CLIPSERV for reply.  As long as the clipboard
           contents still match the request we put in there, we know
           that CLIPSERV hasn't responded yet. Yield() to give
           CLIPSERV a chance. Because 2F/1680 Yield doesn't work so
           good, especially in 3.1 Enhanced mode, Yield (in WINCLIP.C)
           does a few INT 28h Idle calls instead */
        while (strcmp(buf, s = GetClipString()) == 0)
        {
            Yield();
            if (i++ > 10)
                cleanup(``CLIPSERV went down", 1);
            FreeClipString(s);
        }

        /* Finally, the clipboard contents have been changed, hopefully
           by CLIPSERV.  (We could check for the case of other programs
           changing the clipboard at just the wrong moment, but this
           works fine in practice.)  So we display what is hopefully
           CLIPSERV's reply. */
        puts(s);

        FreeClipString(s);
    }

    /* Free up memory, restore the old clipboard contents, and leave */
    cleanup(NULL, 0);
    return 0;
}
```

Figure 10: **The CMDCLIP program is a superior alternative to the batch file approach embodied in Figure 9.**

easiest way is to use the PUTCLIP program, shown in Figure 7, where NOTEPAD is running and where the DOS box has changed its own window title.

The code in the full-blown CLIPSRV1.C very much resembles that in the earlier overview of Figure 2. The key function in this code listing is check_clipboard(), which is called every time the program receives a WM_DRAWCLIPBOARD message. In addition to testing for Clipboard changes from the DOS box and for the CMDCLIP identifier that begins each command/request, check_clipboard() makes a local copy of the Clipboard contents, peels off the identifier, and then uses the remaining string as a parameter to a private message, WM_CLIPCMD.

The function clipcmd() is the handler for these private WM_CLIPCMD messages. It checks for the three built-in commands and dispatches to the appropriate function. For example, RUN calls do_run(), which turns into a WinExec().

To use CLIPSRV1, you can add CLIPSRV1.EXE to the RUN= line in WIN.INI, or run it from Program Manager or another Windows shell. Remember that the program has no window and will appear to do nothing when you run it. Also remember that unless CLIPSRV1 is running, putting a string like

```
CMDCLIP RUN notepad.exe
```

into the Clipboard will do nothing more than put that string into the Clipboard. CLIPSRV1 is required to turn this string into a command.

On the other hand, CLIPSRV1 is *all* that is required. As you can see simply by hanging a few lines of C code off the Clipboard as an odd kind of Clipboard "viewer," we've opened up some of the Windows API to DOS programs.

GETTING BACK RETURN VALUES If you want to give DOS programs access to additional API calls, all you have to do is come up with another command and have it dispatch to the appropriate function. For example, REGCLPFMT might call RegisterClipboardFormat(), and GETVERS might call GetVersion().

This last suggestion discloses a problem: How can the DOS program get the *return values* from these functions? Actually, the problem was present even in CMDCLIP RUN and WinExec(), because the DOS program that currently puts a CMDCLIP RUN into the Clipboard has no way of knowing whether the underlying WinExec() call succeeded. As written, CLIPSRV1 simply throws away the return value from WinExec().

Well, CLIPSRV1 was just an experiment. We now can write the real Clipboard server, CLIPSERV. The first thing to do, then, is to tweak the built-in commands so they can do something intelligent with return values. CLIPSERV should put a return value *back* in the Clipboard as a reply to the DOS program's request. That way, the DOS program can be told whether its request made it over the Windows wall and that the Clipboard server is indeed running.

To accomplish this, Figure 8 shows a changed version of the do_run() function, together with a new function, put_clip_str(). With this code in place, CLIPSERV will respond to a

```
CMDCLIP RUN x
```

by running

```
retval = WinExec("x", SW_NORMAL)
```

CLIPSERV will then place

```
CLIPREPLY RUN
```

followed by a string representation of retval back into the Clipboard.

To retrieve this reply, you could always follow a PUTCLIP CMDCLIP by running the GETCLIP program presented in last issue's Lab Notes. Figure 9 shows how a DOS batch file, CLIPRUN, could do this.

BASHING THE CLIPBOARD CLIPRUN .BAT is not what I would call a triumphant piece of work, though. For one thing, to interpret the rather cryptic results it displays you must know that WinExec() returns a number greater than 32 when it succeeds. Further, CLIPRUN is timing-dependent: If GETCLIP runs before the CLIPSERV has carried out the request, you can get an inappropriate return value in the Clipboard. And, of course, what happens if CLIPSERV isn't running at all?

Thinking about these questions opens up another, equally fundamental problem that you may have been wondering about all along: What happens to the original contents of the Clipboard?

The Clipboard is a great vehicle for sending DOS requests to a Windows server, both because DOS programs have access to the Clipboard via the Enhanced mode INT 2Fh API and because it's easy to write servers that hang off the Clipboard waiting for WM_DRAWCLIPBOARD messages instead of wasting CPU time with polling. The Clipboard has a serious limitation in that there is *only one,* which any program can use whenever it wants. The moral is that whenever an alternative viewer program uses the Clipboard, it should clean up after itself. Once a CLIPSERV transaction is completed, the previous contents of the Clipboard should be restored.

My initial plan was to rely on Richard Hale Shaw's CLIPSTAC program (*PC Magazine,* August 1992) to do this. CLIPSTAC actually provides an API with such messages as WM_CLIPSTACPUSH and WM_CLIPSTACPOP. The saving and restoring of the Clipboard contents, however, should really be done by the DOS program. With CLIPSERV in place, it would be easy for DOS programs to make the necessary SendMessage() calls to communicate with CLIPSTAC. But this presents the same sort of chicken-and-egg problem that we saw with RegisterClipboardFormat(). Basically, whatever underlying mechanism CLIPSERV uses, it needs to be something that doesn't require CLIPSERV!

What's needed instead of CLIPSTAC is a new DOS program that is better than lashing PUTCLIP and GETCLIP together into CLIPRUN.BAT. This DOS program, for which CMDCLIP seems a natural name, uses the WINCLIP library functions—PutClipString() and GetClipString()—that were presented in Part I as the basis of PUTCLIP and GETCLIP. In essence, CMDCLIP:

- Ensures that CLIPSERV is running;
- Saves the current Clipboard contents;
- Puts a request in the Clipboard;
- Waits to get back a reply;
- Displays the reply;
- Restores the saved Clipboard contents.

CMDCLIP.C is shown in Figure 10. The comments in the source code explain how it all works except for the Clipserv() install-check function. CMDCLIP determines whether CLIPSERV is running by applying a kind of distant cousin of the Turing Test. That is to say, Clipserv(), which is part of CMDCLIP.C and is listed in Figure 10, tosses a string into the Clipboard. (The previous contents of the Clipboard have already been saved.) It then checks to see whether an intelligent-looking response appears in the Clipboard shortly thereafter (making it somewhat timing-dependent). If one does, CLIPSERV (or a reasonable imposter) must be running. This requires that CLIPSERV respond appropriately to these install-check messages; this additional code for CLIPSERV is:

```
if (_fstrcmp(cmd, "INSTCHECK")== 0)
 put_clip_str(hwnd, "CLIPREPLY
  INSTOK");
```

Everything is now in place except for the code that makes CLIPSERV truly general purpose. Surprisingly, adding this code—which uses a feature of Windows called runtime dynamic linking—is fairly easy, as I will demonstrate in this column in the next issue. □

ANDREW SCHULMAN IS A WRITER AND ENGINEER AT PHAR LAP SOFTWARE IN CAMBRIDGE, MASSACHUSETTS. HE IS COAUTHOR OF THE BOOK *UNDOCUMENTED DOS* AND OF *UNDOCUMENTED WINDOWS* (FORTHCOMING FROM ADDISON-WESLEY).

Recording and Playing Back MIDI Sequences

BY CHARLES PETZOLD

One of the dangers of being a pioneer is that sometimes you get lost. Programming for the Multimedia Extensions for Microsoft Windows is fairly new, and it's often quite different from normal Windows programming. Because programming guidelines and existing code samples are not always clear, it's possible to make mistakes.

In the December 31, 1991, installment of this column, I described a program called RECORD1 that used the low-level waveform audio functions to record and play back sound. After this program was published, I was politely faulted by some of Microsoft's programmers for several flaws in the code.

PROBLEMS WITH RECORD1 When you use the low-level waveform audio functions to record sound, you pass buffers to the API. While recording, the system fills these buffers with waveform audio data and then passes them back to your program. To play back the sound, you pass filled buffers to the API; your program is notified when the buffers have finished being played.

In RECORD1, I passed the buffers to the API one at a time. I should have used a technique called *double-buffering*. When recording sound, a program should first pass two buffers to the API, then pass a third buffer when the first one is returned to the program. This ensures that there is always a buffer present to receive data. Otherwise, some data may be missed during the time one buffer is returned to the program and the second sent out. Double-buffering is also a good idea when playing back waveform audio sound to prevent any gaps in the playback.

The second problem in RECORD1 is that I frequently (and unnecessarily) called waveInPrepareHeader and waveInUnprepareHeader. As you'll recall, these functions call GlobalPageLock and GlobalPageUnlock for both the WAVEHDR structure and the buffer it references. (The memory for both the structure and buffer must be allocated using GlobalAlloc.) Performing a page lock keeps the memory block at the same physical address in memory and—when Windows is running in 386 Enhanced mode—prevents it from being swapped out to disk. This is necessary for memory that must be accessed by the system at hardware interrupt time.

With the help of a DLL that extends the low-level MIDIAPI, our sample program uses buffered input and output to record and play back MIDI messages.

However, in RECORD1 I called the waveInPrepareHeader and waveInUnprepareHeader functions far too often, adding unnecessary overhead to the time between one buffer being returned to the program and the next being sent out through the API.

I made a third error in RECORD1 when I didn't check whether waveInPrepareHeader or waveOutPrepareHeader succeeded or failed. When Windows is running in 386 Enhanced mode, it makes use of virtual memory. That is, hard disk space is used for swapping code and data segments in and out of real memory. Because of this, there is a large amount of global memory available. If you obtain an error from a GlobalAlloc call, it means you've run out of both physical memory and disk space for swapping.

The success of waveInPrepareHeader and waveOutPrepareHeader, however, depends on the success of GlobalPageLock, which in turn depends on the availability of physical memory installed in the system. Thus, although a GlobalAlloc call may succeed because there is plenty of disk space available, GlobalPageLock can fail due to insufficient physical memory. When you use a lot of page-locked memory, it is just about guaranteed that a GlobalPageLock call will fail long before GlobalAlloc does.

LET'S TRY IT AGAIN I carefully considered these criticisms of RECORD1 while working on the MIDREC ("MIDI Record," shortened to the six-character CompuServe filename maximum) program shown in this column. MIDREC makes use of the MIDBUF dynamic link library that I described in the last column. MIDBUF implements an extension to the Windows multimedia API to allow a program to use buffered input and output to record and play back sequences of MIDI messages. You use this API extension in a very similar manner to the low-level waveform audio functions.

I'll assume here that you have a MIDI board (or a synthesizer board with a MIDI connector box) installed, and that you've installed a Windows driver for this board. I'll also assume that you have a MIDI keyboard (or other MIDI controller), and that you've connected the MIDI Out port of the keyboard to the MIDI In port of the MIDI board.

When you use MIDREC to record,

MIDI messages come from the keyboard and are stored in memory. You can then play back the MIDI sequence on an internal or external synthesizer.

Figures 1 through 6 list the source code for the MIDREC program. You can compile the program from the DOS command line using either the Microsoft C/C++ 7.0 compiler (with the Windows Software Development Kit), or the Borland C++ 3.1 compiler. For the Microsoft compiler, use

```
NMAKE MIDREC.MSC
```

and for the Borland compiler, use

```
MAKE -f MIDREC.BCP
```

Both compilers include all the header files and import libraries necessary for creating Windows programs that take advantage of the Multimedia Extensions. You'll also need the MIDBUF.H header file and MIDBUF.LIB import library for compiling the program, and MIDBUF.DLL for running it. You can download all MIDBUF and MIDREC files from PC MagNet, or you can get them on-disk by sending a postcard with your name and address to PC Magazine, Attention Katherine West, Environments, One Park Avenue, New York, NY 10016. No phone calls, please.

The program's main window has five buttons in two rows: Record and End on the first; Play, Pause, and End on the second. Buttons are enabled only when it is valid to press them. When the program begins execution, only the Record button is enabled. If you press Record, you enable only the first End button. At this point, you can bang away at a MIDI keyboard connected to the MIDI In port of your MIDI board. (Before playing the keyboard, though, it's a good idea to press one of the instrument buttons on the keyboard to store a MIDI Program Change message.) Press End to stop recording.

When recording a MIDI sequence, it's nice to hear what you're playing. Sometimes this is possible, and sometimes it isn't.

After you record a sequence of MIDI messages, both the Record and Play buttons are enabled. You can re-record a MIDI sequence (erasing the first) by pressing Record, or play back the MIDI sequence by pressing Play, which enables the Pause and End buttons. If you press Pause, playback is halted and the button appears with the text Resume. Otherwise, MIDREC plays the MIDI sequence through the synthesizer.

INPUT, MONITOR, AND OUTPUT MIDREC also includes a menu with a single item labeled Device. The Device submenu lists three types of devices—Input, Monitor, and Output. These invoke three additional submenus that list the available MIDI devices installed on the system. These devices are added to the program's menu by MIDREC's AddDevicesToMenu function.

The Input submenu lists all MIDI input devices. The number of devices is obtained from the midiInGetNumDevs function, and the device names are obtained from midiInGetDevCaps. If you have a single MIDI In port on a single MIDI board (the usual case), you'll see just one device on this list. If you have more than one MIDI input device, you can use this menu to select the device from which you wish to record MIDI sequences.

The Output submenu lists all the MIDI output devices, including the MIDI Mapper device (if the MIDI Mapper is installed) as well as the real output devices (the number of which is obtained from midiOutGetNumDevs). Typically, you'll see three devices listed—the MIDI Mapper, the internal synthesizer, and the MIDI Out port of the MIDI board, which may be connected to an external synthesizer. The MIDI Mapper device is checked as a default, but you can select a different output device for playback if you wish.

When recording a MIDI sequence, it's always nice to hear what you're playing. Sometimes it's possible to configure your hardware to allow this, and sometimes it isn't. For example, my external synthe-

MIDREC.MSC
Complete Listing

```
#--------------------------------------------
# MIDREC.MSC make file for Microsoft C 7.0
#--------------------------------------------

midrec.exe : midrec.obj midrec.def midrec.res midbuf.lib
  link midrec,, NUL, /nod midbuf slibcew oldnames libw mmsystem, midrec
  rc -t midrec.res

midrec.obj : midrec.c midrec.h midbuf.h
  cl -c -G2sw -Ow -W3 -Zp -Tp midrec.c

midrec.res : midrec.rc midrec.h
  rc -r midrec.rc
```

Figure 1: **This is the make-file for compiling with Microsoft C/C++ 7.0 using the Windows SDK.**

MIDREC.BCP
Complete Listing

```
#--------------------------------------------
# MIDREC.BCP make file for Borland C++ 3.1
#--------------------------------------------

midrec.exe : midrec.obj midrec.def midrec.res midbuf.lib
   tlink /c /n /Tw /Lc:\borlandc\lib c0ws midrec, midrec, NUL, \
         midbuf import mathws cws, midrec
   rc -t midrec.res

midrec.obj : midrec.c midrec.h midbuf.h
   bcc -c -w-par -P -W -2 midrec.c

midrec.res : midrec.rc midrec.h
   rc -r -ic:\borlandc\include midrec.rc
```

Figure 2: **This is the make-file for use with Borland C++ 3.1.**

MIDREC.C

1 of 3

```
/*---------------------------------------
   MIDREC.C -- MIDI Recorder and Player
               (c) Charles Petzold, 1992
  ---------------------------------------*/

#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <string.h>
#include "midbuf.h"
#include "midrec.h"

#define BUFFER_SIZE 4096      // Should be multiple of 8

BOOL FAR PASCAL _export DlgProc (HWND, UINT, UINT, LONG) ;

static char szAppName [] = "MidRec" ;

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
     {
     FARPROC lpDlgProc ;

     lpDlgProc = MakeProcInstance ((FARPROC) DlgProc, hInstance) ;
     DialogBox (hInstance, szAppName, NULL, lpDlgProc) ;
     FreeProcInstance (lpDlgProc) ;

     return 0 ;
     }

          // Functions to allocate and free MIDIHDR structures and buffers
          // -------------------------------------------------------------

LPMIDIHDR AllocMidiHeader (HANDLE hMidi, LPMIDIHDR pmhRoot)
     {
     LPMIDIHDR pmhNew, pmhNext ;

               // Allocate memory for the new MIDIHDR

     pmhNew = (LPMIDIHDR) GlobalAllocPtr (GHND | GMEM_SHARE, sizeof (MIDIHDR));

     if (pmhNew == NULL)
          return NULL ;

               // Allocate memory for the buffer

     pmhNew->lpData = (LPSTR) GlobalAllocPtr (GHND | GMEM_SHARE, BUFFER_SIZE) ;

     if (pmhNew->lpData == NULL)
          {
          GlobalFreePtr (pmhNew) ;
          return NULL ;
          }

     pmhNew->dwBufferLength = BUFFER_SIZE ;

               // Prepare the header

     if (midiInPrepareHeader (hMidi, pmhNew, sizeof (MIDIHDR)))
          {
          GlobalFreePtr (pmhNew->lpData) ;
          GlobalFreePtr (pmhNew) ;
          return NULL ;
          }

               // Attach new header to end of chain

     if (pmhRoot != NULL)
          {
          pmhNext = pmhRoot ;

          while (pmhNext->dwUser != NULL)
               pmhNext = (LPMIDIHDR) pmhNext->dwUser ;

          pmhNext->dwUser = (DWORD) pmhNew ;
          }

     return pmhNew ;
     }

LPMIDIHDR CleanUpMidiHeaderChain (HANDLE hMidi, LPMIDIHDR pmhRoot)
     {
     LPMIDIHDR pmhCurr, pmhLast, pmhNext, pmhRetn ;

     pmhRetn = pmhRoot ;
     pmhCurr = pmhRoot ;
     pmhLast = NULL ;

     while (pmhCurr != NULL)
          {
          pmhNext = (LPMIDIHDR) pmhCurr->dwUser ;

          if (pmhCurr->dwBytesRecorded == 0)
               {
               midiInUnprepareHeader (hMidi, pmhCurr, sizeof (MIDIHDR)) ;

               GlobalFreePtr (pmhCurr->lpData) ;
               GlobalFreePtr (pmhCurr) ;

               if (pmhCurr == pmhRoot)
                    pmhRetn = NULL ;

               if (pmhLast != NULL)
                    pmhLast->dwUser = (DWORD) pmhNext ;

               pmhCurr = pmhLast ;
               }

          else if (pmhCurr->dwBytesRecorded < BUFFER_SIZE)
               {
               midiInUnprepareHeader (hMidi, pmhCurr, sizeof (MIDIHDR)) ;

               GlobalReAllocPtr (pmhCurr->lpData,
                                 pmhCurr->dwBytesRecorded, 0) ;

               midiInPrepareHeader (hMidi, pmhCurr, sizeof (MIDIHDR)) ;

               pmhCurr->dwBufferLength = pmhCurr->dwBytesRecorded ;
               }

          pmhLast = pmhCurr ;
          pmhCurr = pmhNext ;
          }
     return pmhRetn ;
     }

VOID FreeMidiHeaderChain (HANDLE hMidi, LPMIDIHDR pmhRoot)
     {
     LPMIDIHDR pmhNext, pmhTemp ;

     pmhNext = pmhRoot ;

     while (pmhNext != NULL)
          {
          pmhTemp = (LPMIDIHDR) pmhNext->dwUser ;

          midiInUnprepareHeader (hMidi, pmhNext, sizeof (MIDIHDR)) ;

          GlobalFreePtr (pmhNext->lpData) ;
          GlobalFreePtr (pmhNext) ;

          pmhNext = pmhTemp ;
          }
     }

          // Add MIDI device lists to the program's menu
          // -------------------------------------------

WORD AddDevicesToMenu (HWND hwnd, int iNumInpDevs, int iNumOutDevs)
     {
     HMENU       hMenu, hMenuInp, hMenuMon, hMenuOut ;
     int         i ;
     MIDIINCAPS  mic ;
     MIDIOUTCAPS moc ;
     WORD        wDefaultOut ;

     hMenu = GetMenu (hwnd) ;

               // Create "Input" popup menu

     hMenuInp = CreateMenu () ;

     for (i = 0 ; i < iNumInpDevs ; i++)
          {
          midiInGetDevCaps (i, &mic, sizeof (MIDIINCAPS)) ;
          AppendMenu (hMenuInp, MF_STRING, ID_DEV_INP + i, mic.szPname) ;
          }

     CheckMenuItem (hMenuInp, 0, MF_BYPOSITION | MF_CHECKED) ;
     ModifyMenu (hMenu, ID_DEV_INP, MF_POPUP, hMenuInp, "&Input") ;

               // Create "Monitor" and "Output" popup menus

     hMenuMon = CreateMenu () ;
     hMenuOut = CreateMenu () ;

     AppendMenu (hMenuMon, MF_STRING, ID_DEV_MON, "&None") ;

     if (!midiOutGetDevCaps (MIDIMAPPER, &moc, sizeof (moc)))
          {
          AppendMenu (hMenuMon, MF_STRING, ID_DEV_MON + 1, moc.szPname) ;
          AppendMenu (hMenuOut, MF_STRING, ID_DEV_OUT    , moc.szPname) ;

          wDefaultOut = 0 ;
          }
     else
          wDefaultOut = 1 ;

                         // Add the rest of the MIDI devices

     for (i = 0 ; i < iNumOutDevs ; i++)
          {
          midiOutGetDevCaps (i, &moc, sizeof (moc)) ;
          AppendMenu (hMenuMon, MF_STRING, ID_DEV_MON + i + 2, moc.szPname) ;
          AppendMenu (hMenuOut, MF_STRING, ID_DEV_OUT + i + 1, moc.szPname) ;
          }

     CheckMenuItem (hMenuMon, 0, MF_BYPOSITION | MF_CHECKED) ;
     CheckMenuItem (hMenuOut, 0, MF_BYPOSITION | MF_CHECKED) ;
```

Figure 3: **This listing contains all source code for the program.**

MIDREC.C

2 of 3

```
     ModifyMenu (hMenu, ID_DEV_MON, MF_POPUP, hMenuMon, "&Monitor") ;
     ModifyMenu (hMenu, ID_DEV_OUT, MF_POPUP, hMenuOut, "&Output") ;

     return wDefaultOut ;
     }

BOOL FAR PASCAL _export DlgProc (HWND hwnd, UINT message, UINT wParam,
                                                    LONG lParam)
     {
     static BOOL      bRecording, bPlaying, bEnding, bPaused, bTerminating ;
     static char      szInpError[] = { "Error opening MIDI input port!" } ;
     static char      szOutError[] = { "Error opening MIDI output port!" } ;
     static char      szMonError[] = { "Error opening MIDI output port "
                                       "for monitoring input!  Continuing." } ;
     static char      szMemError[] = { "Error allocating memory!" } ;
     static HMIDIIN   hMidiIn ;
     static HMIDIOUT  hMidiOut ;
     static int       iNumInpDevs, iNumOutDevs ;
     static LPMIDIHDR pMidiHdrRoot, pMidiHdrNext, pMidiHdr ;
     static WORD      wDeviceInp, wDeviceMon, wDeviceOut ;
     HMENU            hMenu ;
     int              i ;

     switch (message)
          {
          case WM_INITDIALOG:

               if (0 == (iNumInpDevs = midiInGetNumDevs ()))
                    {
                    MessageBox (hwnd, "No MIDI Input Devices!", szAppName,
                                MB_ICONEXCLAMATION | MB_OK) ;
                    DestroyWindow (hwnd) ;
                    }

               if (0 == (iNumOutDevs = midiOutGetNumDevs ()))
                    {
                    MessageBox (hwnd, "No MIDI Output Devices!", szAppName,
                                MB_ICONEXCLAMATION | MB_OK) ;
                    DestroyWindow (hwnd) ;
                    }

               wDeviceOut = AddDevicesToMenu (hwnd, iNumInpDevs, iNumOutDevs) ;

               return TRUE ;

          case WM_COMMAND:
               hMenu = GetMenu (hwnd) ;

               switch (wParam)
                    {
                    case ID_RECORD_BEG:

                                   // Open MIDI In port for recording

                         if (midiInOpen (&hMidiIn, wDeviceInp, hwnd, 0L,
                                         CALLBACK_WINDOW))
                              {
                              MessageBox (hwnd, szInpError, szAppName,
                                          MB_ICONEXCLAMATION | MB_OK) ;

                              return TRUE ;
                              }

                                   // Open MIDI Out port for monitoring
                                   //      (continue if unable to open it)

                         if (wDeviceMon > 0)
                              {
                              if (midiOutOpen (&hMidiOut, wDeviceMon - 2,
                                               0L, 0L, 0L))
                                   {
                                   hMidiOut = NULL ;
                                   MessageBox (hwnd, szMonError, szAppName,
                                               MB_ICONEXCLAMATION | MB_OK) ;
                                   }
                              }
                         else
                              hMidiOut = NULL ;

                         return TRUE ;

                    case ID_RECORD_END:
                                   // Reset and close input

                         bEnding = TRUE ;

                         midiInReset (hMidiIn) ;
                         midiInClose (hMidiIn) ;

                         return TRUE ;

                    case ID_PLAY_BEG:
                                   // Open MIDI Out port for playing

                         if (midiOutOpen (&hMidiOut, wDeviceOut - 1,
                                          hwnd, 0L, CALLBACK_WINDOW))
                              {
                              MessageBox (hwnd, szOutError, szAppName,
                                          MB_ICONEXCLAMATION | MB_OK) ;
                              }
                         return TRUE ;

                    case ID_PLAY_PAUSE:
                                   // Pause or restart output

                         if (!bPaused)
                              {
                              midiOutPause (hMidiOut) ;

                                   // All Notes Off message

                              for (i = 0 ; i < 16 ; i++)
                                   midiOutShortMsg (hMidiOut, 0x7BB0 + i) ;

                              SetDlgItemText (hwnd, ID_PLAY_PAUSE, "Resume") ;
                              bPaused = TRUE ;
                              }
                         else
                              {
                              midiOutRestart (hMidiOut) ;
                              SetDlgItemText (hwnd, ID_PLAY_PAUSE, "Pause") ;
                              bPaused = FALSE ;
                              }

                         return TRUE ;

                    case ID_PLAY_END:
                                   // Reset the port and close it

                         bEnding = TRUE ;
                         midiOutReset (hMidiOut) ;
                         midiOutClose (hMidiOut) ;
                         return TRUE ;

                    default:
                         break ;
                    }

               if (wParam >= ID_DEV_INP & wParam < ID_DEV_MON)
                    {
                    CheckMenuItem (hMenu, wDeviceInp + ID_DEV_INP,
                                   MF_UNCHECKED) ;

                    wDeviceInp = wParam - ID_DEV_INP ;

                    CheckMenuItem (hMenu, wDeviceInp + ID_DEV_INP,
                                   MF_CHECKED) ;
                    return 0 ;
                    }

               else if (wParam >= ID_DEV_MON & wParam < ID_DEV_OUT)
                    {
                    CheckMenuItem (hMenu, wDeviceMon + ID_DEV_MON,
                                   MF_UNCHECKED) ;

                    wDeviceMon = wParam - ID_DEV_MON ;

                    CheckMenuItem (hMenu, wDeviceMon + ID_DEV_MON,
                                   MF_CHECKED) ;
                    return 0 ;
                    }

               if (wParam >= ID_DEV_OUT)
                    {
                    CheckMenuItem (hMenu, wDeviceOut + ID_DEV_OUT,
                                   MF_UNCHECKED) ;

                    wDeviceOut = wParam - ID_DEV_OUT ;

                    CheckMenuItem (hMenu, wDeviceOut + ID_DEV_OUT,
                                   MF_CHECKED) ;
                    return 0 ;
                    }

               break ;

          case MM_MIM_OPEN:
               hMidiIn = wParam ;

                         // Free existing headers

               FreeMidiHeaderChain (hMidiIn, pMidiHdrRoot) ;

                         // Allocate root header

               if (NULL == (pMidiHdrRoot = AllocMidiHeader (hMidiIn, NULL)))
                    {
                    midiInClose (hMidiIn) ;
                    MessageBox (hwnd, szMemError, szAppName,
                                MB_ICONEXCLAMATION | MB_OK) ;

                    return TRUE ;
                    }

                         // Allocate next header

               if (NULL == (pMidiHdrNext = AllocMidiHeader (hMidiIn,
                                                            pMidiHdrRoot)))
```

```
            {
            FreeMidiHeaderChain (hMidiIn, pMidiHdrRoot) ;
            midiInClose (hMidiIn) ;
            MessageBox (hwnd, szMemError, szAppName,
                        MB_ICONEXCLAMATION | MB_OK) ;

            return TRUE ;
            }
                // Enable and disable buttons

        EnableWindow (GetDlgItem (hwnd, ID_RECORD_BEG), FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_RECORD_END), TRUE)  ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_BEG),   FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_PAUSE), FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_END),   FALSE) ;
        SetFocus (GetDlgItem (hwnd, ID_RECORD_END)) ;

                // Submit the buffers for receiving data

        midiInShortBuffer (hMidiIn, pMidiHdrRoot, sizeof (MIDIHDR)) ;
        midiInShortBuffer (hMidiIn, pMidiHdrNext, sizeof (MIDIHDR)) ;

                    // Begin recording

        midiInStart (hMidiIn) ;
        bRecording = TRUE ;
        bEnding = FALSE ;
        return TRUE ;

    case MM_MIM_DATA:
        if (hMidiOut)
            {
            midiOutShortMsg (hMidiOut, lParam) ;
            }

        return TRUE ;

    case MM_MIM_LONGDATA:

        if (bEnding)
            return TRUE ;

        pMidiHdrNext = AllocMidiHeader (hMidiIn, pMidiHdrRoot) ;

        if (pMidiHdrNext == NULL)
            {
            midiInReset (hMidiIn) ;
            midiInClose (hMidiIn) ;
            MessageBox (hwnd, szMemError, szAppName,
                        MB_ICONEXCLAMATION | MB_OK) ;

            return TRUE ;
            }

        midiInShortBuffer (hMidiIn, pMidiHdrNext, sizeof (MIDIHDR));

        return TRUE ;

    case MM_MIM_CLOSE:
                    // Close the monitoring output port

        if (hMidiOut)
            {
            midiOutReset (hMidiOut) ;
            midiOutClose (hMidiOut) ;
            }

                    // Enable and Disable Buttons

        EnableWindow (GetDlgItem (hwnd, ID_RECORD_BEG), TRUE) ;
        EnableWindow (GetDlgItem (hwnd, ID_RECORD_END), FALSE) ;
        SetFocus (GetDlgItem (hwnd, ID_RECORD_BEG)) ;

        pMidiHdrRoot = CleanUpMidiHeaderChain (hMidiIn, pMidiHdrRoot) ;

        if (pMidiHdrRoot != NULL)
            {
            EnableWindow (GetDlgItem (hwnd, ID_PLAY_BEG),   TRUE)  ;
            EnableWindow (GetDlgItem (hwnd, ID_PLAY_PAUSE), FALSE) ;
            EnableWindow (GetDlgItem (hwnd, ID_PLAY_END),   FALSE) ;
            SetFocus (GetDlgItem (hwnd, ID_PLAY_BEG)) ;
            }

        bRecording = FALSE ;

        if (bTerminating)
            {
            FreeMidiHeaderChain (hMidiIn, pMidiHdrRoot) ;
            SendMessage (hwnd, WM_SYSCOMMAND, SC_CLOSE, 0L) ;
            }

        return TRUE ;

    case MM_MOM_OPEN:
        hMidiOut = wParam ;

                // Enable and Disable Buttons

        EnableWindow (GetDlgItem (hwnd, ID_RECORD_BEG), FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_RECORD_END), FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_BEG),   FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_PAUSE), TRUE)  ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_END),   TRUE)  ;
        SetFocus (GetDlgItem (hwnd, ID_PLAY_END)) ;

                // Submit the root buffer to begin playing

        midiOutShortBuffer (hMidiOut, pMidiHdrRoot, sizeof (MIDIHDR)) ;

                // If there's a second buffer, submit that also

        if (NULL != (pMidiHdr = (LPMIDIHDR) pMidiHdrRoot->dwUser))
            midiOutShortBuffer (hMidiOut, pMidiHdr, sizeof (MIDIHDR)) ;

        bEnding = FALSE ;
        bPlaying = TRUE ;
        return TRUE ;

    case MM_MOM_DONE:

                // If stopping playback, just return

        if (bEnding)
            return TRUE ;

                // Get header of buffer just finished playing

        pMidiHdr = (LPMIDIHDR) lParam ;

                // Get header of next buffer (already submitted)

        pMidiHdr = (LPMIDIHDR) pMidiHdr->dwUser ;

                // Get header of next buffer to submit now

        if (pMidiHdr != NULL)
            pMidiHdr = (LPMIDIHDR) pMidiHdr->dwUser ;

        if (pMidiHdr != NULL)
            midiOutShortBuffer (hMidiOut, pMidiHdr, sizeof (MIDIHDR)) ;
        else
            {
            midiOutReset (hMidiOut) ;
            midiOutClose (hMidiOut) ;
            }

        return TRUE ;

    case MM_MOM_CLOSE:

                // Enable and Disable Buttons

        EnableWindow (GetDlgItem (hwnd, ID_RECORD_BEG), TRUE)  ;
        EnableWindow (GetDlgItem (hwnd, ID_RECORD_END), TRUE)  ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_BEG),   TRUE)  ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_PAUSE), FALSE) ;
        EnableWindow (GetDlgItem (hwnd, ID_PLAY_END),   FALSE) ;
        SetFocus (GetDlgItem (hwnd, ID_PLAY_BEG)) ;

        SetDlgItemText (hwnd, ID_PLAY_PAUSE, "Pause") ;
        bPaused = FALSE ;
        bPlaying = FALSE ;

        if (bTerminating)
            {
            FreeMidiHeaderChain (hMidiIn, pMidiHdrRoot) ;
            SendMessage (hwnd, WM_SYSCOMMAND, SC_CLOSE, 0L) ;
            }

        return TRUE ;

    case WM_SYSCOMMAND:
        switch (wParam)
            {
            case SC_CLOSE:
                if (bRecording)
                    {
                    bTerminating = TRUE ;
                    bEnding = TRUE ;
                    midiInReset (hMidiIn) ;
                    midiInClose (hMidiIn) ;
                    return TRUE ;
                    }

                if (bPlaying)
                    {
                    bTerminating = TRUE ;
                    bEnding = TRUE ;
                    midiOutReset (hMidiOut) ;
                    midiOutClose (hMidiOut) ;
                    return TRUE ;
                    }

                EndDialog (hwnd, 0) ;
                return TRUE ;
            }
        break ;
    }
return FALSE ;
}
```

MIDI In port in the front of the box and MIDI In, MIDI Thru, and MIDI Out ports in the back. MIDI messages coming into both MIDI In ports are combined to play the synthesizer. The MIDI Thru port is an output port that duplicates MIDI input messages coming into the MIDI In port on the back of the box (but not the one on the front).

Figure 7 shows how a MIDI keyboard, the Roland SC-55, and a MIDI board (such as the Sound Blaster with the MIDI connector box) can be connected so you can use MIDREC to record from the keyboard and play back over the Roland SC-55. While playing the keyboard (with or able to monitor what you're playing on the synthesizer.

Some MIDI boards have their own MIDI Thru ports that duplicate what comes into the board through the MIDI In port. You may need a MIDI mixer to combine the MIDI Thru output and the MIDI Out output to run into an external synthesizer.

If none of these hardware connections are possible, you may be able to monitor your playing through software. This is the purpose of the Monitor submenu, which lists all the MIDI output devices, including an additional option—None—that is checked by default. If you select an output device on the Monitor submenu, MIDREC will attempt to open that device when recording. All MIDI messages coming from the MIDI input device are then sent to the selected MIDI output device.

MIDREC.RC

Complete Listing

```
/*-------------------------
   MIDREC.RC resource script
  -------------------------*/

#include <windows.h>
#include "midrec.h"

MidRec MENU
     {
     POPUP "&Devices"
          {
          MENUITEM "&Input",        ID_DEV_INP
          MENUITEM "&Monitor",      ID_DEV_MON
          MENUITEM "&Output",       ID_DEV_OUT
          }
     }

MidRec DIALOG 32768, 0, 152, 52
     STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX |
WS_VISIBLE
     CAPTION "MIDI Player / Recorder"
     MENU     MidRec
     {
     DEFPUSHBUTTON "Record"  ID_RECORD_BEG,  28,  8, 40, 14
     PUSHBUTTON    "End"     ID_RECORD_END,  76,  8, 40, 14, WS_DISABLED

     PUSHBUTTON    "Play"    ID_PLAY_BEG,     8, 30, 40, 14, WS_DISABLED
     PUSHBUTTON    "Pause"   ID_PLAY_PAUSE,  56, 30, 40, 14, WS_DISABLED
     PUSHBUTTON    "End"     ID_PLAY_END,   104, 30, 40, 14, WS_DISABLED
     }
```

Figure 4: **This resource script defines the menu and the dialog template used for the program's main window.**

MIDREC.DEF

Complete Listing

```
;-----------------------------------
; MIDREC.DEF module definition file
;-----------------------------------

NAME           MIDREC

DESCRIPTION    'MIDI Recorder and Player (c) Charles Petzold, 1992'
EXETYPE        WINDOWS
STUB           'WINSTUB.EXE'
CODE           PRELOAD MOVEABLE DISCARDABLE
DATA           PRELOAD MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      8192
```

Figure 5: **Here is the program's module definition file.**

Complete Listing

```
/*----------------------
   MIDREC.H header file
  ----------------------*/

#define ID_RECORD_BEG     10
#define ID_RECORD_END     11
#define ID_PLAY_BEG       12
#define ID_PLAY_PAUSE     13
#define ID_PLAY_END       14

#define ID_DEV_INP        100
#define ID_DEV_MON        200
#define ID_DEV_OUT        300
```

Figure 6: **This header file defines constants used for the dialog box and menu.**

However, it's not always possible to arbitrarily select a MIDI output port for monitoring. For example, I'm currently using a Sound Blaster board with a MIDI connector box that contains MIDI In and MIDI Out ports. You can't open both of these ports simultaneously. So, if I use this facility, I'm not able to monitor my playing on an external synthesizer. I can, however, simultaneously open the MIDI In port and the internal synthesizer on the Sound Blaster. I can monitor my playing that way and then play it back over an external synthesizer.

At any rate, you may have to do some experimenting. If you need some help, drop a message on the Programming forum on PC MagNet and we'll see if we can work it out.

THE INNER WORKINGS Much of MIDREC's structure is similar to RECORD1's. The program uses a dialog box as its main window. Messages from the buttons and the menu are processed in the DlgProc function.

When you press the Record button, MIDREC opens the selected MIDI input device and—if you've selected an output device for monitoring—the MIDI output device. If this output device cannot be opened, the program displays a message box but goes ahead with the recording operation.

Opening the MIDI input device generates an MM_MIM_OPEN message.

Configuring Your Hardware for Monitoring

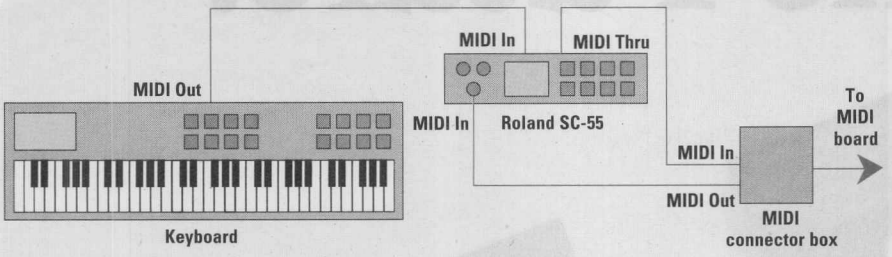


Figure 7: **This illustrates one possible configuration for monitoring what you're playing on the keyboard while also recording it.**

MIDREC handles this message by allocating two MIDIHDR structures and two buffers that are each 4096 bytes long. This memory allocation is performed in the AllocMidiHeader function in MIDREC.C.

As you might recall, the MIDIHDR structure contains a field called dwUser that a program can use for anything it wants. I use this field in order to maintain a linked list of MIDIHDR structures. A pointer to the first allocated MIDIHDR structure is stored in the static variable pMidiHdrRoot in DlgProc. A pointer to the second allocated MIDIHDR structure is stored in the dwUser field of the first structure. At a later time, a pointer to the third MIDIHDR structure will be stored in the dwUser field of the second structure. The AllocMidiHeader function also calls midiInPrepareHeader to lock the structures and buffers in memory.

If this is successful, MIDREC enables and disables the appropriate buttons on the window, and then calls midiInShortBuffer twice. You won't find this function, which allows for buffered MIDI I/O, listed in the *Multimedia Programmer's Reference* manual. It's part of the extensions I added to the multimedia API in the MIDBUF dynamic link library. MIDREC concludes MM_MIM_OPEN processing by calling midiInStart to enable MIDI input.

Even when using MIDBUF for buffered MIDI I/O, a program still gets MM_MIM_DATA messages that contain individual MIDI messages. If a MIDI output port has been opened for monitoring, these MIDI messages are simply passed to the output device by a call to *midiOutShortMsg*.

An MM_MIM_LONGDATA message indicates that a buffer has been filled and is being returned to the program. MIDREC responds by allocating a new buffer and passing it to the API with another call to midiInShortBuffer. This process continues until the program runs out of memory or until the user presses the End button.

MIDREC responds to either case by calling midiInReset and midiInClose,

Multimedia Windows supports MIDI files through the Media Control Interface. But MCI can't record MIDI input into a MIDI file, so that's a chore left for us.

which generates an MM_MIM_CLOSE message, causing MIDREC to call the CleanUpMidiHeaderChain function. This function deletes any MIDIHDR structures in the linked list that reference empty buffers, and reallocates the buffer size for any buffer that's only partially filled. If there are any buffers left after this operation, the Play button is enabled. (All the buffers could be empty if you haven't played anything on the MIDI keyboard, or if the keyboard isn't connected properly.)

Pressing Play causes MIDREC to open the selected MIDI output device, generating an MM_MOM_OPEN message. MIDREC responds by enabling and disabling the appropriate buttons and by passing the first two buffers in the linked list to midiOutShortMessage. This is another of the functions in the extended API that I implemented in the MIDBUF dynamic link library.

As each buffer is finished playing, MIDREC receives an MM_MOM_DONE message, and responds by submitting the next buffer to midiOutShortMessage. If no buffers are left, MIDREC calls midiOutReset and midiOutClose. The appropriate buttons are enabled and disabled during the MM_MOM_CLOSE message.

You can also Pause the playback by making use of the third and fourth new functions, which are implemented in MIDBUF.DLL—midiOutPause and midiOutRestart.

One further note: To simplify global memory allocation under protected mode, I've made use of the GlobalAllocPtr, GlobalFreePtr, and GlobalReAllocPtr macros that are defined in the WINDOWSX.H header file. This header file is included in both the Microsoft Windows SDK and Borland C++, Version 3.1, and defines a number of handy macros. For example, the GlobalAllocPtr macro calls GlobalAlloc and GlobalLock so a program can avoid dealing with global memory handles.

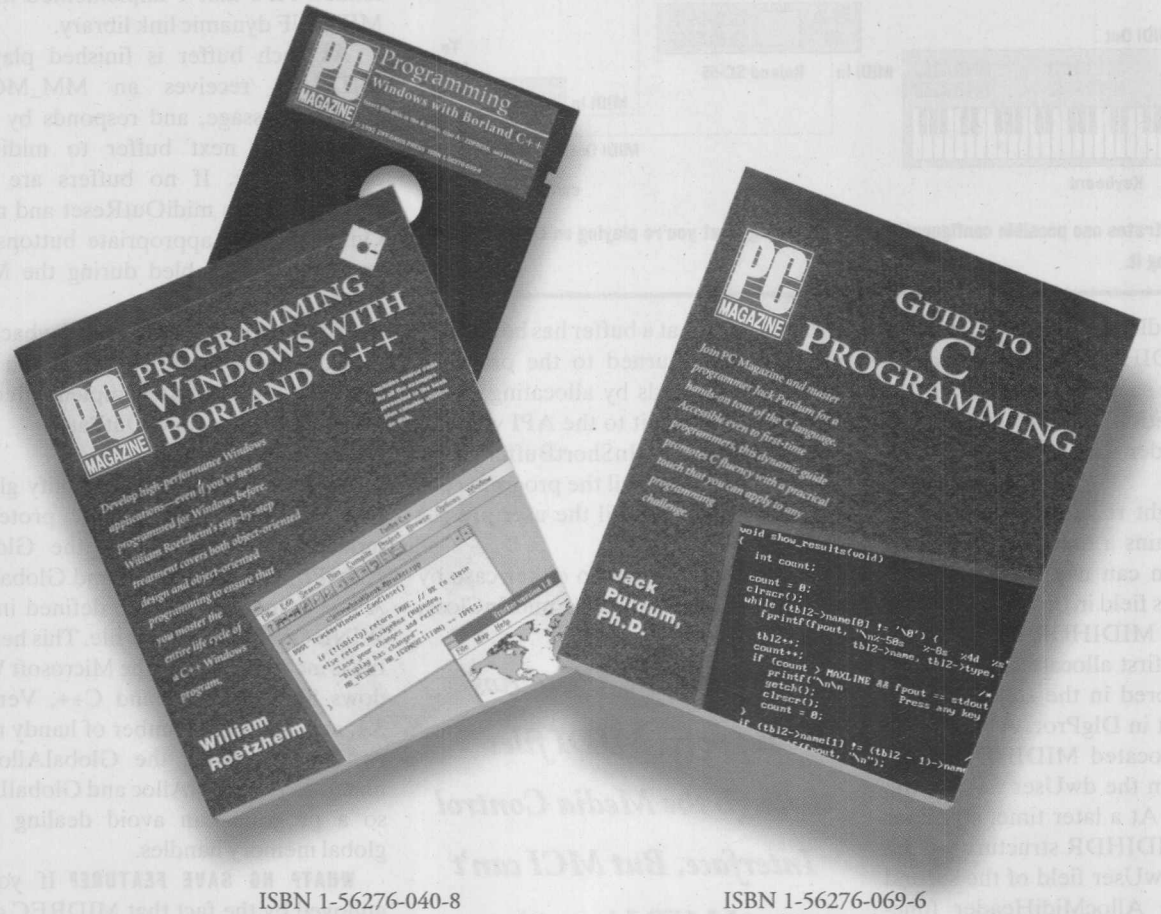
WHAT? NO SAVE FEATURE? If you're annoyed by the fact that MIDREC can't save MIDI sequences to disk and reload them, please be patient. I deliberately resisted adding such a feature to MIDREC because an industry-standard file-format for storing MIDI sequences already exists.

The Multimedia Extensions for Microsoft Windows support MIDI files (with the extension .MID) through the Media Control Interface. Indeed, Microsoft ships Windows 3.1 with a MIDI file that you can play using the Media Player utility. However, MCI cannot record MIDI input into a MIDI file, so that's a chore left for us.

This extended series on the Multimedia Extensions for Windows will culminate with a program that lets you create MIDI files by playing on a MIDI keyboard (or other controller).

Before we get to that, however, we'll need to tackle a couple of preliminaries, including MIDI support under MCI and the MIDI file format. □

C the Future.



ISBN 1-56276-040-8
Price $39.95

ISBN 1-56276-069-6
Price $29.95

C and C++ have become the languages of choice among today's programmers and future programmers. Whether you're just entering the world of C or moving to the power of C++, the authorities at *PC Magazine* have the knowledge to get you where you want to go.

If you're new to C programming or looking to sharpen your skills, **PC Magazine Guide to C Programming** will get you the results that you want. Internationally acclaimed author, and C instructor Jack Purdum provides a solid foundation in all aspects of C programming. Within no time, Dr. Purdum's unique instructional methods will have you writing functional C code that you can apply on any platform using any C compiler.

Windows programming just got easier. World-renowned expert William Roetzheim shows you how to make Windows Graphical User Interface application development a productive and rewarding experience in **PC Magazine Programming Windows with Borland C++**. This unique book/disk package covers object-oriented programming, the Windows Application Program Interface, and the Object Windows Library, to help C programmers into the power and elegance of Borland C++.

Waldensoftware
Waldenbooks

Visit your local Waldensoftware or Waldenbooks store, or call to order 1-800-322-2000. *PC Magazine Guide to C Programming*: Dept. 597, Item #6781. *PC Magazine Programming Windows with Borland C++*: Dept. 597, Item #6862.

ZIFF-DAVIS ZD PRESS

CIRCLE 354 ON READER SERVICE CARD

LAB NOTES

Accessing the Windows API from the DOS Box, Part 3

BY ANDREW SCHULMAN

While Windows limits what a DOS program can do, a DOS program can ask a bona fide Windows program to act in its stead. The trick, however, is in finding a Windows program that will act as a surrogate for a DOS program. In last issue's Lab Notes we started to build such a Windows application, CLIPSERV, which carries out the requests that come from DOS programs and that appear in the Windows Clipboard.

To put the DOS programs' requests into the Windows Clipboard, we also built a companion program, CMDCLIP. CMDCLIP demonstrates how DOS programs can place text into the clipboard by using INT 2Fh functions that were described in Part 1 of this article. The special format CMDCLIP uses to put text into the Clipboard enables CLIPSERV to interpret it as a request to be carried out rather than as plain text.

CMDCLIP and CLIPSERV work as a team, but as written, they're rather specialized. So in this third and final part, I want to show you how to generalize the basic CMDCLIP/CLIPSERV mechanism. This will enable CMDCLIP to call (indirectly) any desired Windows application programming interface (API) function, and it will provide you with a technique you can apply when writing other DOS programs.

RUNTIME DYNAMIC LINKING Because the goal is to provide DOS programs with access to *any* Windows API function (indeed, to any function in a Dynamic Link Library, whether or not it is part of the Windows API), CLIPSERV must be able to link to any DLL function without knowing in advance what that function will be. This requires the use of a documented, but underutilized aspect of Windows called *runtime dynamic linking*. To understand how runtime dynamic linking works, it's good to start by discussing how Windows API functions are normally accessed.

Usually, when a Windows program calls a function such as WinExec(), the call is hard-wired into the program, as shown below:

This Part broadens the scope of CLIPSERV and CMDCLIP so you can call any Windows API or DLL function from a DOS application.

```
/* in WINDOWS.H */
WORD FAR PASCAL WinExec(LPSTR,
  WORD);
/* in the program */
#include "windows.h"
// . . .
WinExec("notepad", SW_NORMAL);
```

In this example, Windows uses *dynamic linking* behind the scenes; that is, Windows links to the appropriate DLL and calls the function at the time the call is performed. But since the header file hard-codes the needed linking information and the main program hard-wires the call to the function, from the program's perspective, the function call doesn't look at all dynamic or flexible.

The inflexibility of this not-so-dynamic linking is usually not noticeable (and doesn't matter), since most programs do know in advance what Windows API function they'll be using. Hard-wired function calls won't do, however, if you want to build a general-purpose program like CLIPSERV, whose job is to call *any* Windows API function on behalf of a DOS program. You could, of course, use brute force to add an element of flexibility by using an `if` statement to select which function to call:

```
if (funcname == "WinExec")
  WinExec(. . .);
else if (funcname == "SendMessage")
  SendMessage(. . .);
else if (funcname == "PostMessage")
  PostMessage(. . .);
else if . . .
```

There are limitations to this approach, though. For one thing, since there are something like 1,200 different Windows API functions, this technique would require a prohibitively large `if` statement! Moreover, your program would work only with whatever functions existed (and that you knew about) at the time you built the program. If a new version of Windows or a new DLL became available, you'd have to modify your program to make use of them. Similarly, if you wanted to call functions from DLLs you created after writing the program, you'd be out of luck.

The litmus test for *truly dynamic* linking, then, is whether a program can call functions that didn't exist (or whose names were unknown) when the program was written. It is vital to CLIPSERV's purpose that it be able to call functions in other DLLs, and advance knowledge

Somehow, a satisfactory CLIPSERV-type program must be able to link to such functions *while it's running.*

Fortunately, Windows provides just the functionality needed to write such a general-purpose program. Last time, in CLIPSRV1, I took strings such as "CMDCLIP RUN" and turned them into WinExec() calls. For example, when CLIPSRV1 is running and a DOS program puts the string "CMDCLIP RUN notepad.exe" into the Windows Clipboard, CLIPSRV1 interprets this as a signal to call

```
WinExec("notepad.exe", SW_NORMAL)
```

In essence, CLIPSRV1 makes an *association* between RUN and WinExec(). Windows has a similar way of associating function names with actual functions built right in. If you have both the name of a Windows API function and the name of the DLL it lives in, you can get a callable pointer to that function. To get it, you first call another function that will return the desired pointer. This second function might, for example, be called GetProc(). Here's an example of how GetProc() would be used:

```
WORD (FAR PASCAL *WinExec)(LPSTR,
 WORD);
WinExec = GetProc("KERNEL",
 "WINEXEC");
if (WinExec != 0)
  (*WinExec)("notepad", SW_NORMAL);
else
  fail("Can't find WinExec
    function!");
```

In the above code `WinExec` is a function pointer: It holds the address of the WinExec() function in memory. Since `(*f)(x)` and `f(x)` are equivalent in ANSI C, the way the function is called isn't all that different from the way you'd normally call WinExec. What's different is how the program gets WinExec's address in the first place.

Ordinarily, a program relies on Windows to take care of this when the program is loaded; that is, before the program even starts running. This "normal" form of dynamic linking in Windows is called *load-time dynamic linking.* In the code excerpt above, however, the program does the dynamic linking. Since this takes place while the program is running, it is called *runtime dynamic linking.*

GetProc() is not part of the Windows API. As shown in Figure 1, it is built by using the Windows API functions GetModuleHandle(), LoadLibrary(), and GetProcAddress(). GetProc() first tries to use GetModuleHandle() to turn a module name (such as KERNEL) into a module handle. If that fails (for example, if the named module hasn't been loaded yet), GetProc() uses LoadLibrary(). Once it has a module handle, GetProc() passes the handle, along with the name of the function (such as WINEXEC), to the GetProcAddress() function. GetProcAddress() then returns a far pointer to the function.

Windows applications that use a macro language (like Microsoft Word for Windows) will almost always contain a statement (for example, `Declare`) that lets you call Windows API functions and functions in other DLLs from the macro language. For instance, in a Lab Notes on Word Basic (*PC Magazine,* October 29, 1991), I showed the following code to readers:

```
Declare Sub MyShell Lib "kernel" \
  (lpCmdStr$, nCmdShow As Integer) \
  Alias "WinExec"
Sub MAIN
  MyShell "notepad", 1
End Sub
```

Now that you know about runtime dynamic linking, it becomes clear that WinWord is internally using a function very much like GetProc() to implement the Declare Sub statement. In the example above, the string "kernel" gets passed as the first argument, and the string "WinExec" gets passed as the second argument.

Thus, if we build something like GetProc() into CLIPSERV, add some way to handle parameters to the functions, and give CLIPSERV a new DYNLINK keyword in addition to the RUN, SETTITLE, and EXIT keywords it already knows about, the result will be that we can let a DOS by simply passing in its string name a some representation of the function's arguments. An example might look like this:

```
CMDCLIP DYNLINK kernel winexec
 notepad 1
```

Since almost all functions that a program is likely to want will be in one of three core Windows DLLs—KERNEL, USER, or GDI—we can do better than this. CLIPSERV could look in those three places automatically without the DOS program having to specify those modules. However, since the DOS program might want to call other DLLs, it's a bit faster if the DOS program *does* specify the module. We can allow that by adding some syntactic sugar that lets us specify a function in a particular DLL by prefixing the function name with the DLL name and an exclamation point. Thus, the strings that CLIPSERV accepts will actually look like this:

```
CMDCLIP DYNLINK winexec notepad 1
```

or:

```
CMDCLIP DYNLINK kernel!winexec
 "notepad" 1
```

HANDLING FUNCTION-PARAMETER TYPES

In addition to showing how CLIPSERV expects to receive function names, these examples also give you some idea of how it wants to be told about function arguments. First of all, a string such as "notepad" can simply be typed in, either as is or enclosed within quotation marks. The quotation marks are required if the string contains any spaces. Consider this:

```
CMDCLIP DYNLINK winexec
"notepad hello.txt" 1
```

GetProc()

```
FARPROC GetProc(char *modname, char *funcname)
{
    WORD hModule;
    if (! (hModule = GetModuleHandle(modname)))
        if (! (hModule = LoadLibrary(modname)))
            return 0;
    return GetProcAddress(hModule, funcname);
}
```

Figure 1: **The GetProc() procedure uses runtime dynamic linking in Windows.**

But how about the number 1, the second argument being passed to WinExec()? In a Windows program, a normal call would look something like this:

```
WinExec("notepad", SW_NORMAL);
```

CLIPSERV doesn't "know" about such constants since they are not built into Windows the same way as the string "WINEXEC" is. Constants are in a Windows SDK include file, WINDOWS.H. This means you must look up constants such as SW_NORMAL and type in the equivalent raw number—a messy business. (I'll deal with this problem later.)

In any case, we can see that CLIPSERV must be able to take a string such as "1" and figure out that it should be passed to WinExec(), not as a string, but as a 2-byte integer. CLIPSERV does this by including a type() function, which uses some fairly simple but effective rules to determine the type of an argument. The code for type() is shown in Figure 2. The result is that CLIPSERV can properly call SendMessage(hWnd, msg, wParam, lParam) on the basis of these instructions:

```
CMDCLIP DYNLINK sendmessage 0x1234
 0x0C 0 "hello world"
```

In this illustration, 0x1234 is a hypothetical window handle (HWND). (I'll show you below how the DOS program would get that HWND handle in the first place.) 0x0C is the message number for WM_SETTEXT. 0 is the unused wParam, and "hello world" is the lParam. The result is that the specified window's text will be set, by a DOS program, to "hello world".

PUSHING ARGUMENTS When it receives a string on the Windows Clipboard, CLIPSERV has to do more than figure out what function to call and what type its arguments are. CLIPSERV actually has to *pass* all the arguments to the function. Because CLIPSERV is a general-purpose program, it not only has no idea what function it will be asked to call; it also has no idea how many arguments the function expects or what type they will be. (Nor does it know about the function's return value, but I'll discuss this later.) Somehow CLIPSERV needs a very generic way of making function calls. The same program has to be able to handle this:

```
CMDCLIP DYNLINK winexec notepad 1
```

or this:

```
CMDCLIP DYNLINK sendmessage 0x1234
 0x0C "hello world" 0L
```

Every Windows programmer has probably heard of GENERIC.C, the "mother all Windows programs" that comes with the SDK. Well, CLIPSERV *really is* generic! When it receives a string such as the one shown above on the Windows Clipboard, CLIPSERV must:

- Call GetProc() to get a function pointer to SendMessage() in USER;
- Push the two-byte integer 0x1234, which SendMessage() will interpret as the HWND of the window to be sent this message, on the stack;
- Push the integer 0x0C, representing the WM_SETTEXT message, on the stack;
- Push the integer 0 (unused wParam) on the stack;
- Push a 4-byte far pointer to the string "hello world" on the stack, as the lParam;
- With its arguments now on the stack, call the SendMessage() function pointer;
- Take SendMessage()'s 2-byte return value from the AX register, turn it into a NULL-terminated ASCII string, and put it into the Clipboard.

To accomplish all this requires adding to CLIPSERV what is, essentially, a tiny interpreter that uses a "push loop" to turn string arguments from the Clipboard into actual arguments on the function's stack. The interpreter code is contained in a file, DYNLINK.C, that gets linked with CLIPSERV.C. Unfortunately, even a tiny interpreter is too large to list here in its entirety, though it is presented in outline form in Figure 3. As usual, however, you can download the source code for DYNLINK.C (and any other source and corresponding executables presented in this series) from PC MagNet, archived as CLIP3.ZIP. If you don't have access to PC MagNet, you can receive it by mail. Just send a postcard with your name, address, and preferred disk size to the attention of Katherine West, Lab Notes, PC Magazine, One Park Ave., New York, NY 10016.

In summary, then, CLIPSERV thus comes to consist of the old CLIPSRV1 code plus a runtime dynamic linking routine similar to GetProc() in Figure 1, the type() function in Figure 2, and its interpreter, DYNLINK.

It's worth looking in detail at the "push loop" part of the code from DYNLINK.C, which is shown in Figure 4. For each argument, the push loop uses the type() function from Figure 2 to determine the type of an argument. The switch statement in the PUSH_ARG() macro determines, for a given type, *what* should be pushed. If the argument is of typ_string, then a 4-byte pointer to the string is pushed on the stack. If the argument is of typ_word, it is converted into

type()

```
/*
type() uses some dumb rules to determine the type of an argument:
      if first character of arg is a digit or '-'
            and if arg contains '.' then it's a floating-point number
            else if last character is an 'L' then it's a long
            else it's a unsigned word
      else if first character is an apostrophe
            it's a single-byte character
      otherwise
            it's a string
            if the first char
*/

typedef enum { typ_string, typ_byte, typ_word, typ_long, typ_float,
    typ_buffer, typ_hwnd } TYPE;

TYPE type(char *arg)
{
    if (isdigit(arg[0]) || (arg[0] == '-' && isdigit(arg[1])))
    {
        char *p = arg;
        while (*p)
            if (*p++ == '.')
                return typ_float;
        return (*--p == 'L') ? typ_long : typ_word;
    }
    else if (strcmp(arg, "@buf") == 0)
        return typ_buffer;  // push far ptr to 1k buffer
    else if (strcmp(arg, "@hwnd") == 0)
        return typ_hwnd;    // push caller's HWND
    else
        return (arg[0] == '\'') ? typ_byte : typ_string;
}
```

Figure 2: **CLIPSERV uses this type( ) function to determine what type an argument belongs to.**

a 2-byte number, and that is pushed. (The conversion is handled by axtol(), a function that converts a decimal- or hex-formatted string such as "1234" or "0x1234" or "1234:5678" into an actual 4-byte long number.) The push() function relies on a lovely trick that I learned from a book on OS/2 programming by David Cortesi: If you use the Pascal calling convention, the C function push() pushes its argument, of whatever size, on the stack and leaves it there.

GETTING RETURN VALUES The arguments to the function are now on the stack. CLIPSERV must call the function, get its return value, and put this return value someplace where the DOS program can get at it. Getting return values is easy in programs that know in advance what functions they're calling. In this generic program, however, we have no idea what the function will return: It could be an integer, a long, a string, or a floating-point number.

CLIPSERV must put the return value into the Windows Clipboard in the form of a string so the DOS program can access it. But what should be the *format* of this string representation? The DOS program might want to have some say in how the string is formatted, for instance whether a numeric return value should be turned into a decimal or into a hexadecimal string.

The answer to both problems is to let the DOS program specify both the size and format for the return value by using an optional printf() mask. For example, if you call the Windows GetVersion() function from the CMDCLIP command line without specifying a return value, it comes back as a 2-byte number formatted with the printf mask 0x%04, which is CLIPSERV's default.

CLIPSERV's default is:

```
C:\PCMAG>cmdclip dynlink getversion
0x0a03
```

The 0x0a03 means that Windows 3.1 is the operating environment (0x0a is, of course, hexadecimal for the number 10). If you want that to be formatted a little differently, you can specify an explicit mask:

```
C:\PCMAG>cmdclip dynlink getversion
%04X
0A03
```

In Windows 3.1, GetVersion() actually returns a 4-byte number, with the Windows version number found at one end and the DOS version at the other. You can use the printf mask to print out all four bytes:

dynlink

```
dynlink(char *clipboard_contents)
{
    FARPROC funcptr;
    int argc;
    char argv[MAX_ARGS];

    argc = split(clipboard_contents, argv); // tokenize string

    use argv[1] to get function pointer:
    if (argv[1] contains '!')
        use explicit module name
    else
        try first USER, then KERNEL, then GDI
    funcptr = GetProc(modname, funcname);

    use optional printf mask for size and format of return value:
    if (argv[argc-1] contains %)
        retval_mask = argv[--argc]

    push loop:
    for each argument (order depends on Pascal vs. cdecl)
        switch (type(args))
            case typ_string:    push((char far *) arg); break;
            case typ_word:      push((unsigned) axtol(arg)); break;
            case typ_long:      push(axtol(arg)); break;
            ...

    arguments are already on the stack -- call the function:
    (*funcptr)();

    get return value:
    switch (retval_type from retval_mask)
        case typ_string: wsprintf(buf, retval_mask, DX:AX); break;
        case typ_word:   wsprintf(buf, retval_mask, AX); break;
        case typ_long:   wsprintf(buf, retval_mask, DX:AX); break;
        ...

    put return value into clipboard:
    put_clip_str(buf);
}
```

Figure 3: **This listing presents a pseudocode outline for the runtime dynamic-linking interpreter used in CLIPSERV.**

DYNLINK.C

Partial Listing

```
/* push(): a trick that relies on pascal calling convention */
void pascal push() { }

// push args on stack, count # of words
#define PUSH_ARG(arg)   \
{   \
    switch (type(arg))  \
    {   \
            case typ_buffer: push((char far *) the_buffer); c += 2; break;  \
            case typ_hwnd:     push(hwnd);                    c += 1;
break;  \
        case typ_string: push((char far *) arg);        c += 2; break;  \
        case typ_byte:   push(arg[1]);                  c += 1; break;  \
        case typ_word:   push((unsigned) axtol(arg));   c += 1; break;  \
        case typ_long:   push(axtol(arg));              c += 2; break;  \
        case typ_float:  push(atof(arg));                 c += 4; break;  \
    }   \
}

// ...

/* push_loop */
if (is_cdecl)
{
      /* push in reverse order for cdecl */
      for (i=argc-1, c=0; i>=start_arg; i--)
            PUSH_ARG(argv[i]);
}
else
{
      for (i=start_arg, c=0; i<argc; i++)
            PUSH_ARG(argv[i]);
}

// ...

/* Convert a string to a long number:  accepts decimal,
   hex, or seg:ofs far pointer */
unsigned long axtol(char *s)
{
      unsigned long ret;
    if (s[0]=='0' && s[1]=='x')
    {
        sscanf(s+2, "%lx", &ret);
        return ret;
    }
    else if (strchr(s, ':'))
      {
            sscanf(s, "%Fp", &ret);
            return ret;
      }
      else
        return atol(s);
}
```

Figure 4: **This partial listing of DYNLINK.C shows the code used to push arguments on the stack.**

```
C:\PCMAG>cmdclip dynlink getversion
 %08lX
05000A03
```

This reply shows that CMDCLIP was running under DOS 5.0 and Windows 3.1. It doesn't display it very well, but remember that CMDCLIP is just a user-interface on top of the PutClipString() and GetClipString() functions. Programs other than CMDCLIP can be built to talk to CLIPSERV, and they can format the return value from GetVersion() or from any other Windows API function, however they want.

Figure 5 shows how the code that processes these printf() masks works. Using

```
"cmdclip dynlink getversion %08lX"
```

as an example, the printf mask is %08lX, and the retval_type() returns that this is typ_long. Thus the switch statement shown in Figure 5 casts the generic function pointer `f`—which by now holds a pointer to GetVersion()—to a LONGFN and makes the actual call (the whole point of this program!) inside a wsprintf() call like so:

```
case typ_long:
  wsprintf(buf, mask,
    ((LONGFN) f)());
```

GetVersion() expects no parameters, so it isn't a very realistic example. However, even a function that *did* expect parameters would be called in the same way, for the parameters are already on the stack, where the push loop put them.

GETTING USEFUL PARAMETERS There is still a big question about how you get useful parameters to Windows API functions when you're running in the DOS box. Earlier I used SendMessage() as an example, and I just made up the window handle (HWND) 0x1234. The question is, where would you get a real window handle when you're running in the DOS box?

From a Windows API call, of course! In the three illustrative command lines below, I'll use CMDCLIP

- to launch the Window Clock,
- to find its window handle (using the FindWindow() function and the Clock window _class name_), and then use that window handle
- to change Clock's title bar to "It's lunch time!":

```
C:\PCMAG>cmdclip dynlink winexec
 clock 1
0x10c6
C:\PCMAG>cmdclip dynlink findwindow
 "Clock" 0L
0x19a8
C:\PCMAG>cmdclip dynlink
 setwindowtext 0x19a8 "It's lunch
 time!"
0x0000
```

What if a program running in the DOS box wants to get its *own* window handle? One way would be to use the CMDCLIP equivalent of FindWindow("tty", 0L), since "tty" is the window class name for DOS boxes. Thus,

```
C:\PCMAG>cmdclip dynlink findwindow
 tty
0L
```

However, if there was more than one DOS box running, this might return the handle to a different DOS box. A better idea, therefore, is to use the @hwnd variable built into CLIPSERV. Specifying @hwnd will cause CLIPSERV to use the window handle of whoever made the dynlink request. Thus,

```
C:\PCMAG>cmdclip dynlink
 setwindowtext @hwnd "Hello world"
```

A similar variable is @buf, which turns into a 4-byte far pointer to a 1K buffer built into CLIPSERV. You can use @buf whenever a function needs to "side effect" a block of memory. You might want to get your window's title, using the Windows GetWindowText() function. This function expects as one of its parameters a far pointer to a buffer. To get a buffer you could, of course, call GlobalAlloc() via CMDCLIP. However, CLIPSERV makes life even easier for its clients by giving them a ready-to-use 1K block of memory. The command line would read:

DYNLINK.C
Partial Listing

```
typedef unsigned (far *FN)();
typedef char far * (far *STRFN)();
typedef char (far *BYTEFN)();
typedef unsigned (far *WORDFN)();
typedef unsigned long (far *LONGFN)();
typedef double (far pascal *FLOATFN)();

static char buf[128];
FN f;
TYPE retval_typ = typ_word;

// ... f = (FN) GetProc(modname, funcname) ...

/* handle optional printf mask */
if (strchr(argv[argc-1], '%'))
      retval_typ = retval_type(mask = argv[--argc]);

// ... push loop here ...

/* args are on the stack : call (*f)() and print retval */
switch (retval_typ)
{
      case typ_string: wsprintf(buf, mask, ((STRFN) f)()); break;
      case typ_byte:   wsprintf(buf, mask, ((BYTEFN) f)()); break;
      case typ_word:   wsprintf(buf, mask, f()); break;
      case typ_long:   wsprintf(buf, mask, ((LONGFN) f)()); break;
      case typ_float:  wsprintf(buf, mask, ((FLOATFN) f)()); break;
}
```

```
// ...

put_clip_str(buf);

// ...

/* retval_type() uses a printf() mask (e.g., %s or %lX)
   to determine the return value's type */
TYPE retval_type(char *s)
{
    while (*s)
    {
        switch (*s)
        {
            case 's' :  return typ_string; break;      // %s or %Fs
            case 'c' :  return typ_byte; break;        // %c
            case 'p' : case 'l' : case 'I' : case 'O' : case 'U' :
                       return typ_long; break;         // %lX, etc.
            case 'e' : case 'E' : case 'f' : case 'g' : case 'G' :
                       return typ_float; break;        // %f, etc.
        }
        s++;
    }

    /* still here */
    return typ_word;    // %u, %d, %x, etc.
}
```

Figure 5: **This excerpt contains the DYNLINK.C code that handles return values and actually calls the function.**

```
C:\PCMAG>cmdclip dynlink
 getwindowtext @hwnd @buf 128
```

REMOTE PROCEDURE CALLING CMDCLIP shows how Windows API calls can be provided on the DOS side of the fence. It is little more than a demonstration, however, of such remote procedure calling. When writing DOS programs you would probably want to call these functions from your own program, not from a mini-interpreter operated from the command line.

To call a Windows function such as WinExec() from a DOS program, you need to have CLIPSERV on the Windows side of the fence. The other thing you need is a *stub function,* whose job is to look as much as possible like the Windows function that you actually want to call; and to take care of marshalling its arguments to send to CLIPSERV.

Figure 6 provides DOS remote-procedure call versions of WinExec() and SendMessage() as illustrative examples. Calling these functions from a DOS program looks much like calling them from a Windows program. The only difference is that they take a little longer to execute, because, under the hood, the functions are furiously busy passing their parameters to CLIPSERV and getting back their return values via the Clipboard.

There is actually one other difference in the SendMessage() function: The DOS version takes an additional argument. SendMessage() is sometimes called with an lParam holding a far pointer to a string; at other times, the lParam could also hold a 4-byte number. Any DOS program that calls this version of SendMessage() is running in a different address space from the genuine SendMessage() function in Windows. This means that the DOS program's far pointer would be meaningless to SendMessage().

Since the DOS version of SendMessage() can't send over a far pointer, it has to send over the string itself. This is what I meant, in a comment in last issue's Lab Notes, about "flattening" pointers. The inability to pass pointers to remote functions is one obstacle to achieving transparency; that is, of making the interface to a DOS version of SendMessage() look exactly like the Windows version. Still, it's close enough. After all, a while ago we couldn't call SendMessage() from DOS at all, and now we're quibbling over an extra parameter! This is progress.

THE PROBLEMS WITH REMOTE CALLS There are other obstacles to making Windows API calls from the DOS box look exactly like Windows API calls from a Windows program. For one, if you issue a bogus API call from your DOS program, you must remember that it's CLIPSERV that is actually making the call on your behalf. Windows doesn't know about surrogates or servers, and though the fault is yours (as it were), if something is wrong with the way you called the function, CLIPSERV may experience a UAE or GP (general protection) fault and be shut down.

Help in avoiding such unacceptable shutdowns is available from the Windows ToolHelp library. This library comes with the Windows 3.1 SDK, but it can also run on top of 3.0 and software developers are allowed to redistribute the DLL with their programs for the benefit of 3.0 users. The Windows ToolHelp library contains an extremely useful function called InterruptRegister(), which lets Windows programs install a callback function to handle their own GP faults and UAEs. By using InterruptRegister() to watch for these faults (remember, a GP fault is just an INT 0Dh), CLIPSERV will *not* go down because of an error in the way you called a Windows API function. Figure 7 shows how an InterruptRegister handler can be used in conjunction with the Catch() and Throw() functions, which are Windows equivalents of the C setjmp() and longjmp() functions.

Another problem: Either CMDCLIP or any other DOS program that talks to CLIPSERV needs to loop to get its return value from CLIPSERV. There is ju no equivalent to the nice WM_DRAW CLIPBOARD wake-up call that CLIPSERV receives. (Windows has its advantages!) In CMDCLIP, I get the return value by calling the Yield() function in a loop. This is not only inelegant, but it seems to disclose a problem with the implementation of the INT 2Fh AX=1680h Yield call in Windows 3.1. The problem

WinExec() and SendMessage()
Partial Listing

(a)

```
typedef int BOOL;
typedef unsigned short WORD;
typedef WORD HWND;
typedef unsigned long DWORD;
typedef char far *LPSTR;

#define SW_NORMAL       1

WORD WinExec(LPSTR cmd, WORD sw)
{
    char buf[128];
    char *s;
    WORD ret;
    sprintf(buf, "CMDCLIP DYNLINK KERNEL!WINEXEC \"%Fs\" %d",
        cmd, sw);
    PutClipString(buf);
    Yield();      // should be loop: see cmdclip.c

    s = GetClipString();
    sscanf(s, "%x", &ret);
    FreeClipString(s);
    return ret;
}
```

(b)

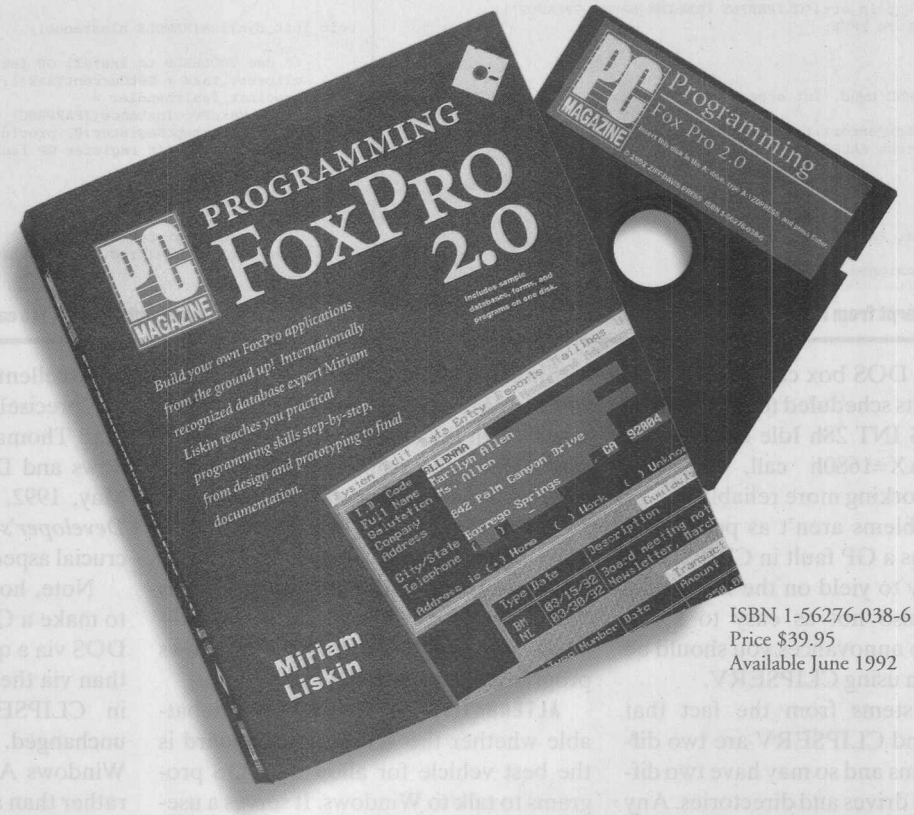
```
#define WM_SETTEXT 0x0c

DWORD SendMessage(HWND hwnd, WORD msg, WORD wparam,
                  DWORD lparam, BOOL str)
{
    char buf[256];
    DWORD ret;
    if (str)
    {
        sprintf(buf, "CMDCLIP DYNLINK USER!SENDMESSAGE "
                     "0x%04x 0x%04x 0x%04x \"%Fs\"",
            hwnd, msg, wparam, (LPSTR) lparam);
    }
    else
    {
        sprintf(buf, "CMDCLIP DYNLINK USER!SENDMESSAGE "
                     "0x%04x 0x%04x 0x%04x 0x%08lxL",
            hwnd, msg, wparam, lparam);
    }
    PutClipString(buf);
      Yield();    // should be loop: see cmdclip.c

    s = GetClipString(buf);
    sscanf(s, "%lx", &ret);
    FreeClipString(s);
    return ret;
}
```

Figure 6: **These two code excerpts show (a) a DOS version of WinExec( ) that looks like the Windows API WinExec( ) call, but which internally takes care of packaging up its arguments to send to CLIPSERV, and (b) a similar DOS version of SendMessage( ).**

Miriam Liskin, PC Magazine, and FoxPro 2.0. Three Pros, No Cons.



ISBN 1-56276-038-6
Price $39.95
Available June 1992

Miriam Liskin, *PC Magazine*, and FoxPro 2.0. They're all recognized as the very best at what they do. Now they have teamed up to deliver *PC Magazine Programming FoxPro 2.0*, one exhaustive volume that covers the FoxPro language itself and the practical skills you need to take your applications from first vision to finished product. Sensible design, realistic prototyping, effective documentation—all presented in the comprehensive style that has established Liskin as the premier authority in database instruction. Liskin focuses on the development of typical business applications from the ground up, building your base of programming knowledge and showing you how to put all the pieces together. Concepts and techniques are explained in plain English, and are backed up by one disk full of sample forms, programs, and databases. The disk will save you hours of input and debugging time, while offering you a vast store of code that you can adapt to your own application designs.

Whatever your past programming experience, you'll gain confidence fast with the comprehensive coverage and real-world examples in *PC Magazine Programming FoxPro 2.0*.

ZIFF-DAVIS ZD PRESS

Waldensoftware®
Waldenbooks®

Visit your local Waldensoftware or Waldenbooks store, or call to order 1-800-322-2000. Dept. 594, Item #6282. Check your yellow pages for the store nearest you.

CIRCLE 354 ON READER SERVICE CARD

DYNLINK.C

Partial Listing

```
static BOOL in_dynlink = 0;
static FARPROC procinst_faulthandler;
static CATCHBUF catchbuf = {0} ;
static HANDLE clipserv_task = 0;

BOOL dynlink_error(void)
{
      if (Catch(catchbuf) == 0)
            return FALSE;
      else
      {
            put_clip_str("CLIPREPLY DYNLINK ERROR GPFAULT");
            return TRUE;
      }
}

BOOL dynlink(HWND hwnd, int argc, char *argv[])
{
      if (dynlink_error())    // catch: come back here on fault
            return FALSE;

      // ...
}

void _export far FaultHandler(void)
{
      static unsigned intnum;

      _asm mov ax, word ptr [bp+8]
      _asm mov intnum, ax

      if ((in_dynlink) &&
            (intnum == 0x0d) &&
            (GetCurrentTask() == clipserv_task))
            Throw(catchbuf, 1);
      else
            return;
}

void init_dynlink(HANDLE hInstance)
{
      /* use TOOLHELP to install GP fault handler for DYNLINK */
      clipserv_task = GetCurrentTask();
      procinst_faulthandler =
            MakeProcInstance((FARPROC) FaultHandler, hInstance);
      if (! InterruptRegister(0, procinst_faulthandler))
            fail("Can't register GP fault handler!");
}

void fini_dynlink(void)
{
      InterruptUnRegister(0);
      FreeProcInstance(procinst_faulthandler);
}
```

Figure 7: **This excerpt from DYNLINK.C contains the GP fault-handling code. A Windows program can use InterruptRegister() to catch its own UAEs.**

is that once a DOS box calls Yield, it almost never gets scheduled to run! By substituting DOS INT 28h Idle calls for the INT 2Fh AX=1680h call, I've got CMDCLIP working more reliably.

Other problems aren't as potentially catastrophic as a GP fault in CLIPSERV or an inability to yield on the DOS side, but they're also not as easy to solve. There are two annoyances you should be aware of when using CLIPSERV.

The first stems from the fact that CMDCLIP and CLIPSERV are two different programs and so may have two different current drives and directories. Any API calls that involve paths will need to specify the *entire* absolute path; relative paths depend on both programs having the same current drive and directory.

Second, I have said that CLIPSERV can give a DOS program access to *any* Windows API function. As currently written, this isn't quite true. Specifically, CLIPSERV doesn't handle Windows functions—such as InterruptRegister()—that expect to have pointers to callback functions passed to them. An improved version of CLIPSERV would need to provide either special-case handling for a few of the more useful functions with callbacks, or it would need a mechanism for calling such functions as the Windows Switch VM and the Callback function (INT 2Fh AX=1685h) in the DOS program's virtual machine. CLIPSERV would need to supply surrogates for these callbacks; in this version, it doesn't.

Similarly, while the DOS program can call SendMessage(), it can't *receive* messages. With the exception of function return values, the communication is one-way. In the event-driven Windows environment, this is a real handicap. On the other hand, if you really want to be writing full-blown, event-driven programs that can receive messages and install callbacks, you should be writing a Windows program, not a hybrid DOS program.

ALTERNATIVE APPROACHES It's debatable whether the Windows Clipboard is the best vehicle for allowing DOS programs to talk to Windows. It serves a useful purpose here because it is the easiest route to take and because it helps to introduce many of the key issues involved in making DOS and Windows programs talk to each other. But the Clipboard certainly isn't the best way to go.

An infinitely superior (but also more difficult) route would be to write a 32-bit Windows virtual device driver (VxD) to manage a queue or pipe that DOS and Windows programs could share. This would involve a third program (a .386 file) in addition to CMDCLIP and CLIPSERV. VxDs can provide their own APIs both to real-mode DOS and to protected-mode Windows programs. A DOS program somewhat like CMDCLIP (you might call it CMDQ) could use the VxD to put requests into a queue rather than into the Clipboard. And a Windows program like CLIPSERV (called QSERV) could get requests from this queue. For an excellent introduction to using VxDs for precisely this purpose, you should read Thomas W. Olsen's "Making Windows and DOS Programs Talk," in the May, 1992, issue of the *Windows/DOS Developer's Journal.* VxDs really are a crucial aspect of Windows programming.

Note, however, that if you do decide to make a QSERV program that talks to DOS via a queue or pipe in a VxD rather than via the clipboard, much of the code in CLIPSERV can be carried over unchanged. You might decide to pass Windows API calls over in binary form rather than as text, and you might change other things about the program, but many issues would remain the same.

In the course of building the GETCLIP, PUTCLIP, CMDCLIP, and CLIPSERV utilities, I've tried to show you how to access the Windows Clipboard from the DOS box, how to write event-driven message handlers in Windows without using a switch statement, how to use Windows' runtime dynamic linking, and how to catch your own GP faults. In fact, writing CLIPSERV was really an exercise in using various features of Windows to write very generic, general-purpose software. □

ANDREW SCHULMAN IS A WRITER AND ENGINEER AT PHAR LAP SOFTWARE IN CAMBRIDGE, MASSACHUSETTS. HE IS COAUTHOR OF THE BOOK *UNDOCUMENTED DOS* AND OF *UNDOCUMENTED WINDOWS*, FORTHCOMING FROM ADDISON-WESLEY.